4

BBN Systems and Technologies Corporation

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 7144

A GUIDE TO IRUS-II APPLICATION DEVELOPMENT

Damris Ayuso, Gerard Donlon, Dawn MacLaughlin, Lance Ramshaw, Philip Resnik, Varda Shaked, Ralph Weischedel

September 1989

Submitted by:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, MA 02138

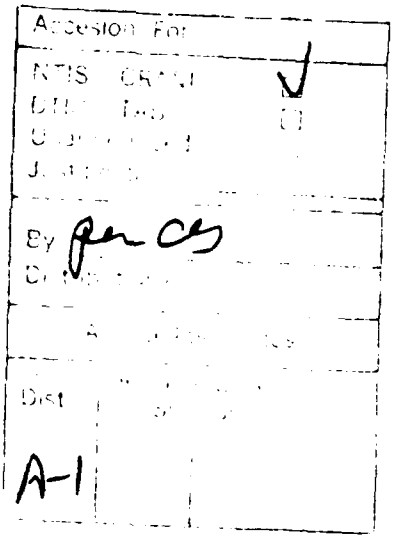
Submitted to:

Defense Advanced Research Projects Agency (DARPA)
1400 Wilson Blvd.
Arlington, VA 22209

Report No. 7144

A GUIDE TO IRUS-II APPLICATION DEVELOPMENT

Damris Ayuso, Gerard Donlon, Dawn MacLaughlin, Lance Ramshaw, Philip Resnik, Varda Shaked, Ralph Weischedel



September 1989

Submitted by:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, MA 02138

Submitted to:

Defense Advanced Research Projects Agency (DARPA)
1400 Wilson Blvd.
Arlington, VA 22209

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by ONR under Contract No. 00014-85-C-0016. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 7144	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION BBN Systems and Technologies Corporation	6b. OFFICE SYMBOL (If applicable)
	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cmbridge, MA 02138	7b. ADDRESS (City, State, and ZIP Code) Department of the Navy Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Defense Advanced Research Projects Agency	8b. OFFICE SYMBOL (If applicable) DARPA ISTO
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F00014-85-C-0016
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A Guide to Irus-II Application Development

12. PERSONAL AUTHOR(S)
Ralph Weischedel, Damaris Ayuso, Gerard Donlon, Dawn MacLaughlin, Lance Ramshaw, Philip Resnik, Varda Shaked

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim Technical Rept	FROM _____ TO _____	1989, September	88

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Janus, Irus, Natural Language Interfaces, User Interfaces, Knowledge Bases.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

IRUS-II is the understanding subsystem of the Janus natural language interface. IRUS-II is a natural language understanding (NLU) shell. That is, it contains domain-independent algorithms, a large grammar of English, domain-independent semantic interpretation rules, and a domain-independent discourse component. In addition, several software aids are provided to customize the system to particular application domains. These software aids output the four knowledge bases necessary for IRUS-II to correctly interpret English utterances and generate appropriate code for simultaneous access to multiple application systems.

This document describes the four knowledge bases and how to construct them.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

Table of Contents

1. Preface 1

2. Introduction 3

2.1 BBN Natural Language systems - Overview 3
2.1.1 Introduction and Background 3
2.1.2 IRUS-II Components 3
2.1.3 The Evolutionary Step from IRUS to IRUS-II 5
2.2 IRUS-II Capabilities 7
2.2.1 Linguistic Capabilities 7
2.2.2 Handling Ambiguitites 8
2.2.3 Handling Language Peculiarities and Anomalies 9
2.2.4 Mapping to Multiple Underlying Systems 10
2.2.5 Mixed Modality Interaction 10

3. Building an IRUS-II Domain Model 11

3.1 Introduction 11
3.2 Domain Models 12
3.3 Building Domain Models with NIKL 13
3.3.1 NIKL Concepts 14
3.3.2 NIKL Roles and Role Restrictions 14
3.4 NIKL Syntax 15
3.4.1 DEFCONCEPT 15
3.4.2 DEFROLE 16
3.5 KREME 17
3.6 A Brief Domain Model Example 17

4. Semantic Interpretation Rules 21

4.1 Pattern Section 22
4.1.1 Clause IRules 22
4.1.2 NP IRules 24
4.1.3 Adjective IRules 25
4.1.4 Prepositional Phrase IRules 26
4.1.5 Selectional Restriction Tests 27
4.2 Action Section 28
4.2.1 Variable Binding List 28
4.2.2 IRule Body 29
4.2.3 IRule Examples 30
4.3 Creating IRules 32
4.3.1 Using IRACQ 32
4.3.2 An IRACQ session 33
4.3.3 Debugging environment 34
4.3.4 Editing an IRule 35

5. The IRUS Dictionary 37

5.1 Introduction 37
5.2 Add Word 39
5.2.1 Nouns 40
5.2.2 Adjectives 41
5.2.3 Adverbs 41
5.2.4 Verbs 42
5.2.5 Other Categories 45
5.3 Edit Word 46
5.4 The Compounds, Multiples, and Substitute Properties 47
5.4.1 Compounds 48
5.4.2 Multiples 48
5.4.3 Substitute 48
5.5 Print Word 48
5.6 Dictionary Commands 49
5.6.1 Save Dictionary 49
5.6.2 Make Pretty Dictionary 49
5.7 Semantics 49

6. World Model Language Representations of Queries 51

6.1 Intensional Logic 51
6.2 Domain Model Constants and Predicates in WML 52
6.3 Example 52
6.3.1 WML Types 53
6.3.2 Meaningful WML Expressions 54

7. Users' Guide to KNACQ in IRUS-II 57

7.1 Purpose of KNACQ 57
7.2 Adding KNACQ Data to the Domain Model 58
7.2.1 Terms for Concepts 60
7.2.2 Terms for Attribute Roles 60
7.2.3 Caseframe Patterns for Roles 61
7.2.4 Gradeable Adjective Terms 62
7.3 Making KNACQ Data Available in IRUS-II 62

8. Access to Multiple Underlying Systems in Janus 63

8.1 Introduction 63
8.2 The Type System 63
8.3 Normalizing WML Expressions 64
8.3.1 Disjunctive Normal Form 65
8.3.2 System-independent Rewrites 66
8.3.3 Printfunctions 67
8.4 Servers and Services 68
8.5 Execution 70
8.6 An Example Backend 71

APPENDIX A. Installation Instructions 73

A.1 Systems Included in IRUS-II 73
A.2 Installation Instructions 74
A.2.1 Loading IDS Files From FEP Tape(s) 74
A.2.2 Do You Have The Correct IDS Files? 74
A.2.3 Resetting the FEP 75
A.2.4 Do You Have The Correct Microcode? 75
A.2.5 Reading FEP Tapes 75
A.2.6 Editing The Boot Files 76
A.2.7 Booting the BBN NL System 76
A.3 Loading Carry Tapes 76
A.4 Transferring FEP Files Onto Other Machines 77
A.5 Loading a Distribution Tape 77

1. Preface

The purpose of this document is to provide an overview of IRUS-II and assist developers interested in adapting IRUS-II to new application domains. Chapter 2 provides a general introduction and overview. Chapter 3 describes the use of domain models in IRUS-II: it provides suggestions for the development of new domain models and a description of the syntax for the development of domain models with NIKL. Chapter 4 describes the Semantic Interpretation Rules of IRUS-II. Chapter 5 presents the categories and features of the IRUS-II dictionary. Chapter 6 describes the World Model Language (WML), the logical representation language of IRUS-II. Chapter 7 describes using a new knowledge acquisition tool that speeds development of syntactic and semantic information for the dictionary. Chapter 8 describes the process of translating a WML expression into a form that one or more underlying systems can execute. Finally, Appendix A provides instructions for loading the IRUS-II software.

This document supercedes an earlier version, BBN Report 6989.

2. Introduction

2.1 BBN Natural Language systems - Overview

2.1.1 Introduction and Background

BBN has developed an evolutionary sequence of state-of-the-art natural language interfaces as part of DARPA's Strategic Computing Program. IRUS [5, 30], the first in that sequence, was the result of years of DARPA-funded research. IRUS was installed and demonstrated at CINCPACFLT in 1986 as part of the Fleet Command Center Battle Management Program. It was integrated with the FRESH environment at CINCPACFLT as a natural language interface to Navy databases (DBs) and to a mapping and report generation system (OSGP). The initial version of the second evolutionary natural language system, IRUS-II, was demonstrated in May 1987 as part of the Janus system. The 1987 version of Janus integrated understanding capabilities developed by BBN with a generation system developed by ISI.

The current version of the Janus system was first demonstrated in March 1988 as a seamless system integrating language, menus, maps, tables and pointing. Janus now incorporates the IRUS-II understanding system and the SPOKESMAN natural language generation system [13, 15], used for paraphrasing the natural language input, generating English responses to queries, and clarifying ambiguous or partially understood inputs.

A selected bibliography of papers and reports on BBN's natural language projects is provided at the end of this document.

2.1.2 IRUS-II Components

In IRUS-II (as in IRUS), the natural language capability is distinguished both from the conceptual view of the domain and from the details of the particular expert system or application program being used.

2.1.2.1 Front End Components of IRUS-II

The major domain-independent front end modules of the IRUS-II system include a parser and associated grammar, a semantic interpreter, and a discourse subsystem for resolving anaphora and ellipsis. These modules function to parse an English query or command into meaningful phrases and structures, which are translated into a formal semantic representation which represents the meaning of English sentences in a higher order intensional logic called World Model Language (WML) (cf. [11] and chapter 6). WML is essentially independent of the target system's command language and its constants are concepts from a domain model (chapter 3). The domain model contains information about concepts and relations between concepts in a specific domain. IRUS-II currently uses the NIKL [16] knowledge representation formalism to represent the domain model.

Additional front end knowledge bases contribute to the syntactic, semantic, and discourse processing of English queries. A lexicon (chapter 5) contains the information about word and phrase substitutes, parts of speech and syntactic, semantic, and morphological features needed for parsing. A set of semantic interpretation rules (chapter 4) specify mappings between English constructs and concepts and relations in the application domain. These are used to determine whether the incremental syntactic structures produced by the parser are interpretable structures and to specify their interpretation. These rules for generating WML depend on information contained in the domain model. The discourse component uses a set of rules which generate discourse entities based on the structure of the WML, and a separate set of co-reference constraint rules for intra-sentential anaphora resolution.

2.1.2.2 Back End Components of IRUS-II

The task of the back end components of IRUS-II is to take a WML expression representing the meaning of a query and compute the correct command or set of commands to one or more underlying systems, obtaining the result requested by the user. This problem is decomposed into the following steps:

- Simplify the WML expression where possible
- Identify the underlying system(s) which may contribute to the satisfaction of the user's request, making use of descriptions of underlying system capabilities
- Formulate a sequence of calls to the underlying system(s), attempting to minimize cost/effort in instances where system capabilities overlap
- Execute the call sequence
- Generate a response which will be *helpful and informative* for the user

Chapter 8 explains these steps in more detail. The major modules in the backend correspond to these steps and include a module which applies translation rules to an expression and simplifies the result, a module that deduces which underlying system(s) should be utilized, a module that constructs the appropriate calling sequence, and a module that executes the code and makes sense of the results.

Although the constants in WML are concepts from the domain model, the constants in the executable code are determined by the structure of the site's underlying systems. This means that the English-to-WML and simplification translations are domain-dependent but underlying-system independent, while the generation of an executable sequence is both domain-dependent and underlying-system dependent. Therefore, were a database administrator to reorganize the database, the WML and its corresponding simplifications would be unaffected, but it would be necessary to update the appropriate underlying system description.

2.1.3 The Evolutionary Step from IRUS to IRUS-II

The main differences between IRUS and IRUS-II can be characterized as:

1. Using intensional logic (IL) to represent natural language expressions.
2. A sophisticated back end component that maps intensional expressions into their extensions in the underlying systems, incorporating a clear methodology which uses declarative knowledge about underlying system capabilities.
3. A new discourse component which generates discourse entities ("things one can refer to") from the meaning representation of a sentence, and handles references introduced by pointing gestures in a manner consistent with the treatment of linguistic (i.e. pronoun) references.
4. The incorporation of KNACQ [33], with its general rules for treating attributes (chapter 7).

2.1.3.1 Intensional Logic

One of the difficulties in developing a system of logical representation for natural language utterances is that traditional first order logic (e.g. the MRL representation of IRUS) is more limited in its expressive range than natural language is. Intensional logic (e.g. the WML representation of IRUS-II) represents one attempt to broaden the expressive range of logic so that it covers a broader scope of natural language. Intensional logic can represent expressions whose value (extension) varies depending on time (a characteristic typical of real world applications), as in "What were Frederick's last 3 positions?". It can also represent propositions which are either true in some possible world, or necessarily true in all possible worlds, which allows for representing the semantics of hypothetical, or "what-if" situations, as in "Suppose Frederick were C3", as well as modality (e.g. possibility, necessity), as in "Can the unit arrive here within 20 hours?".

WML now allows us to represent predicates or operators that take intensional arguments, such as CHANGE or REQUEST. We now represent the contribution that tense provides to the determination of an evaluation time index, by using operators such as PAST and FUTURE. A uniform treatment of plurals was also established so that singular nouns generate the standard interpretation, and plurals generate a POWER operator acting on a set, allowing for collective as well as distributive readings [21, 22].

In order to generate WML forms, the semantic component of IRUS was modified. The clear interface between separate syntactic and semantic components allowed for changes to the semantic interpreter without perturbing other modules. A new operator, FULL-INTERP is now allowed in IRules to indicate that a constituent should be taken intensionally. While some of the default IRules, one dealing with tense, for example, had to be modified, the IRules and lexical entries that were domain dependent remained exactly the same, reminding us of the advantage of encoding knowledge declaratively.

WML is a strongly typed logical language. The formal syntax and set-theoretic semantics of WML are described in [11]. Further discussion of WML is in Chapter 6.

The World Model Language (WML) representation of a query represents the most abstract stage in the processing of a query by IRUS-II. While the generation of WML is dependent on the syntax and semantics of the

natural language query, the generation of a query in the language of the underlying system from the WML representation is dependent on the interface to the underlying system, the functionality of that system, and the structure of its database or knowledge base.

The WML expression produced by the front end is the input to the back end (to be translated into executable code) and the input to the Paraphraser (to paraphrase the system interpretation of the input).

2.1.3.2 A Better Handling of Mapping to Multiple Underlying Systems

The previous generation of IRUS (e.g., as used in the IRUS/OSGP demonstration system) made use of a limited number of underlying systems, choosing amongst them via a small set of key words embedded in the representation of a query. The IRUS-II natural language interface makes the integration of multiple underlying systems transparent not only to the user (as in the IRUS system) but to much of the natural language system as well. The multiple underlying systems component takes an application-system independent representation of the utterance (the WML expression in a simplified form) and determines which subexpressions fall within the capabilities of which application systems.

The multiple systems module embodies a methodology for combining any number of application systems. No assumptions are made about the existence or absence of overlapping data, etc. Underlying system capabilities are represented declaratively, and a decision procedure which seeks to minimize cost is used to resolve cases in which more than one system is capable of handling a single subexpression.

2.1.3.3 A New Discourse Component

A new discourse component was built for IRUS-II which centers around two main notions:

- The use of a general representation for "communicative acts", or events that happen during the user-machine interactions. These include inputs by the user, pointing actions by the user, system responses, and other system-initiated communications.
- The use of discourse entities [31, 32] as the fundamental unit for representing items that are introduced in the discourse and which may be referred to. Discourse entities may be introduced by an utterance (a "linguistic communicative act"), or by the system (e.g., an object in a table response can "answer communicative act"). The role of discourse entities in Janus is discussed in [3].

The representation of discourse state in Janus now consists of a stack of communicative act structures, with the most recent on top, and a structure representing the focus space.

Each communicative act contains focusing information (a "focus-info" structure) which represents how the communicative act affected the items in focus. We use centering algorithms [8, 7] to track the movement of focus. Each focus-info structure contains fields for the "back center", "forward centers", and "center movement status," e.g., continuing or retaining. The focusing information is used in resolving external anaphora, where the possible referents are taken from the list of forward centers. A linguistic communicative act (representing a user's input) in addition contains fields for the parse tree, WML, discourse entities, and anaphor resolution information for that input.

The separate representation of a focus space stack [9] is meant to contain focusing information regarding a segment of a hierarchical representation of discourse. Since currently we have a flat representation of discourse state, the focus space stack merely contains the focus-info for each communicative act.

Treatment of intra-sentential anaphora was also added to the discourse component. Syntactic constraints as represented in the c-command rules of Reinhart [19] are stated declaratively as rules and applied in determining possible antecedents.

When more than one possible referent is found for an anaphor, the options are presented to the user, who indicates which to use.

2.2 IRUS-II Capabilities

2.2.1 Linguistic Capabilities

IRUS-II handles a very wide range of English structures, including the following:

- **Pronouns.** The IRUS-II system allows many common uses of "it", "that", "he", etc. to refer to items that have been introduced in the discourse, i.e., discourse entities introduced by previous queries or pointing actions. It also understands pronouns referring to items *within* the sentence (intra-sententially). For example, "Did Frederick report *its* readiness as C1?"
- **Referring Expressions.** The IRUS-II system will allow common use of phrases such as "that VP squadron", "those ships", etc. to refer to items in previous queries.
- **Ellipsis.** People do not always use complete sentences to express their thoughts. If IRUS-II finds an utterance that is not a complete sentence but is a complete noun phrase or prepositional phrase, it will attempt to understand it elliptically by looking back at the previous query for a corresponding NP or PP constituent that has a similar semantic type, and substituting this new utterance for it. For example, after "Show the TAGOS", the input "Squadrons", will be interpreted to mean "Show the squadrons". However, the approach is limited because the substitution is of full constituents only, which may not lead to the intended reading. In the following elliptical query sequence: "Which C1 carriers are in port?" "Subs?" it is possible that the user wants the set of submarines to be limited to those that are C1, but IRUS-II would substitute the noun phrase "submarines" for the phrase "C1 carriers", rather than just for the noun "carriers", and the answer would include all submarines in port, rather than just those that are C1. The correct decision in such cases depends on semantics in subtle ways. Here is an example where the modifier probably should not be included in the query built from the ellipsis: "Which C1 ships are harpoon capable?" "C2 subs?" IRUS-II currently only tries for interpretations by substituting complete constituents. A paraphrase of the elliptical query will show how the ellipsis was resolved.
- **Quantification.** Words like "all", "any", "some" are handled.
- **Partitives.** Partitions of sets such as "3 of the <x>" or "5 of the fastest <x>"are handled.
- **Conjunction Disjunction.** IRUS covers cases of conjunction where the conjoined constituents are either complete clauses (e.g. "<Display the TAGOS> and <run the SPA model>"), complete noun phrases (e.g. "Show <the VP squadrons> and <the surface ships>"), or complete prepositional phrases (e.g. "Which ships are M1 <on ASW> or <on AAW>?"). It does not in general cover the more complicated syntactic cases where the conjuncts depend on each other by sharing elements (e.g. "List

and display the C1 carriers" "List the TAGOS that were C2 but are now C1"). There is often more than one possible interpretation of a conjunction, and people rely on semantic and pragmatic information in understanding. Thus, most people would assume that the speaker of the following sentence, "List the carriers in Hawaii and San Diego", intended to get a combined list of the carriers that either are in Hawaii or in San Diego, as if it were "list the carriers in Hawaii and [list the carriers] in San Diego." However, handling this style of conjunction in a natural language interface is a research issue, since it requires realizing that "and" functions as an "or". Postmodifiers are usually grouped with the closest possible constituent. Thus "List the frigates and carriers in the IO", would usually be interpreted first as "List <the frigates> and <the carriers in the IO>" listing all the frigates everywhere. A special provision exists in IRUS-II for noun phrases, however, so that if the previous conjuncts have not been postmodified, a postmodifier on the final conjunct will be distributed, yielding "List <the frigates [in the IO]> and <the carriers [in the IO]>".

- **Negation.** For example, "Which subs are not C1?" (Quantification and negation together often causes ambiguity).
- **Comparisons and Superlatives.** IRUS-II handles common comparative forms such as "Is Frederick faster than Spica?", as well as simple superlative constructs such as "Which is the fastest carrier in the Indian Ocean?".
- **Units and Amounts.** e.g., "3 knots", "50 mph".
- **Times and Dates.** These are understood in a wide variety of formats, such as "1900Z", "May 31, 1988", "31/5/88", and "251631May88".
- **Numerical and Distance Calculations.** e.g., "Total the number of planes on all carriers", "How far is Frederick from Hawaii?", "List the TAGOs within 20 miles of SPA 2"
- **Hypothetical World queries.** IRUS-II is being extended to understand hypotheticals like "What will be the arrival time of the unit if it needs to refuel once?"

2.2.2 Handling Ambiguitites

Understanding English in light of ambiguity is a central research problem. In this section, we describe the various kinds of ambiguity and the cases handled in IRUS-II. There are at least seven kinds of ambiguities that may occur in natural language. These are:

1. Semantic Ambiguity - for example, the query "When will we see California?" could refer to either to the nuclear surface ship California or California State

2. Structural Ambiguity - In the query "Display the frigates and carriers that are C1", "C1" may be modifying either "carriers" or "frigates and carriers."

3. Ambiguity in Referring Expressions - after the query "Which TAGOS are assigned to SPA 1 and SPA 2" the command "List their locations" could refer either to the locations of the "TAGOS" or to the locations of "SPA 1 and SPA 2".

4. Conjunction and Disjunction Ambiguity of AND/OR - Aside from the above modifier distribution ambiguity, conjunctions can introduce additional ambiguity - that of AND/OR interpretation. The query "Which SPAs are alerted and within 50 miles of Hawaii" could refer either to "SPAs that are both alerted and are within 50 miles of Hawaii", or to "SPAs that are either alerted or within 50 miles of Hawaii"

6. Quantifier Scope Ambiguity - Sentences containing several quantified noun phrases often cause ambiguities. For example, the sentence "Did all subs in this task force report a readiness problem?", could mean either "Did all subs in this task force report *the same* readiness problem?", or "Did each sub in this task force report *at least one* readiness problem (but the problem may differ from one sub to another)?

7. Collective/Distributive Ambiguity - For example, the query "Are Frederick and Spica communicating?" can be interpreted as "Are Frederick and Spica communicating with each other?" or as "Is Frederick communicating <with something> and is Spica communicating <with something>" These interpretations arise because of the collective reading property that the verb "communicate" possesses.

Currently, the IRUS-II system can completely identify and present to the user (in paraphrasing mode) only semantic ambiguity stemming from more than one word sense. Multiple WMLs are generated in this case.

Structural ambiguity is detected by the parser. However, only the first structural interpretation passing semantic tests is presented to the user. The mechanisms for requesting another parse exist, but have not been incorporated into the user interface. A call to (next-WML) by a system developer obtains the next syntactic reading.

Referring expression ambiguity is detected and the options are presented to the user.

Ambiguities regarding conjunction/disjunction and quantifier scope are not detected. However, for these cases we may "envision" additional interpretations although they have not been actually produced by the Parser, and propose those interpretations (perhaps via a Paraphraser) to the user. For example, in cases of conjunctions that are premodified (Type 3), we can "envision" (or "derive") the WML that corresponds to the premodifier distribution, from the existing WML that corresponds to the interpretation in which the premodifier is attached only to the first NP (and vice versa).

Collective/distributive ambiguity is currently handled by having plurals always generate collective readings; simplification of the WML expression by the back-end includes performing distribution when appropriate (that is, when the predicate expects individuals, but -- because of a plural -- appears to be applying to a set).

2.2.3 Handling Language Peculiarities and Anomalies

Ill-formedness: The IRUS-II system can handle certain types of sentences that are not fully grammatical. The parser only searches for an ill-formed interpretation if it is not able to find a normal one. The classes of ill-formed input currently handled are:

1. omitted articles, e.g., "List units in Pacific".
2. missing prepositions, e.g., "List units Pacific."
3. subject/verb disagreement, e.g., "Which battle group of carriers are in the Pacific?"
4. determiner/number disagreement, e.g., "Is a units near Hawaii?", and
5. incorrect pronoun case, e.g., "Display the unfriendly units between it and they."

Spelling Correction: A spelling corrector is invoked whenever IRUS-II cannot make sense of a word that the user has typed (i.e., the word the user has typed is not in the dictionary and it does not appear to be an inflected form of a word in the dictionary). The spelling corrector proposes a possible list of re-spelled words from which the user can choose the desired word

2.2.4 Mapping to Multiple Underlying Systems

One of the important features of the IRUS-II system is that it can provide access to multuple underlying application systems at a single installation. This access is transparent to the user and has been made transparent to much of the natural language system as well. This process was briefly discussed in section 2.1.3.2.

A paradigmatic case that demonstrates accessing multiple systems, is the CINCPACFLT installation of the IRUS-86 system, in which both a Navy database (IDB) and a Mapping system (OSGP) are accessed. Each has its own data files, with the files of IDB being the more extensive. The interface to those systems was effectively combined in an integrated natural language interface that was able to transfer data found only in IDB to make displays on OSGP.

2.2.5 Mixed Modality Interaction

The natural language mode of input can be integrated with other kinds of input such as graphics. Specifically, when demonstrated as part of Janus, an integrated system that included a map with mousable icons, IRUS-II provided a means for mouse clicks on map icons to be remembered as entities which could then be referred to by deictic expressions in natural language input. For example, one could click on three different submarines and then type in, "Assign those to SPA 2. This kind of mixed-mode interaction is an important part of the seamless character of an interface, allowing the user to move fluidly between modalities, so that the graphical display reflects the result of understanding and responding to the natural language query, and the query in turn can refer to graphical entities.

3. Building an IRUS-II Domain Model

3.1 Introduction

The domain model provides a link between the user's view of the domain and the underlying system's representation of that domain. Once the system has interpreted the user's query in terms of a logical representation that uses entities from the domain model, translation rules are used to convert that logical representation into the query language of the underlying system. This approach effectively isolates the user from the technical details of the underlying system and the formal query language of the underlying structure.

Since the first implementation of IRUS-II was designed to support database queries, the remainder of this chapter will assume a database query application. Regardless of the functionality of the underlying system, the requirements for developing a domain model are similar. Additionally, applications other than database queries may require the implementation of new operators that were not in the first implementation of IRUS-II, such as "WHAT-IF" for hypothetical queries posed to expert systems.

A useful distinction that can be drawn in the analysis of a natural language input utterance is the distinction between the *denotative* aspect of the utterance and its *performative* aspect. The denotative aspect refers to the objects in the domain that are referred to by the utterance. The performative aspect refers to the action that the system is being instructed to perform in regard to those objects. For example, consider the utterance:

```
Display the locations of the C1 units.
```

Display represents the performative aspect of this utterance. The interpretation of *Display* depends on the functionality of the underlying system. The remainder of the sentence (its denotative aspect), on the other hand, can be intepreted independently of the functionality of the system. In one implementation, *Display* may result in a tabular listing; in another implementation it may result in a graphic presentation of a map. But the logical representation of the denotative aspect of the utterance is independent of system functionality.

While additions to the performative support provided by IRUS-II currently require the expertise of the IRUS-II developers, tools are available to make the addition of new denotative support (i.e., a new domain model) a much simpler matter.

3.2 Domain Models

A domain model is a conceptual description of a subset of the entities in the world. The relevant subset is determined by the application of the system. For example, a system that will be used to access a database containing information on a Navy unit's equipment inventory will contain references to items of Navy equipment -- weapons, ships, ports, helicopters, etc., and to relevant data about that equipment -- location, maintenance status, model, identification number, cost, etc.

The entities in a domain model are concepts and roles. Each concept represents a class of things, and typically is named for this class of things. Roles represent relationships in which those concepts participate. For example, the concept SHIP represents the class of things that are ships. Associated with it may be several roles such as SHIP-NAME, ID-NUMBER, COMMANDER, HOME-PORT. For each of these roles, the class of things that are ships are the *domain*. Each of these roles also has a *range* -- the class of things into which they map. For example, the range of SHIP-NAME could be the class STRING, the range of ID-NUMBER could be the class NUMBER, the range of COMMANDER could be the class PERSON (or OFFICER), and the range of HOME-PORT could be the class PORT.

Once the domain developer has identified a concept that should be included in the domain model, then a number of roles for which that concept is the domain will immediately come to mind. In creating those roles, the domain developer will have to specify the classes which are their ranges. These ranges represent additional concepts which will have to be included in the domain model. These concepts, in turn, will be the domains of new roles. Once begun, the process of developing a domain model tends to proceed spontaneously. During implementation, testing, and debugging, the decisions made in the initial design stage will probably have to be altered and adjusted.

The domain developer can approach the problem of identifying these roles and concepts from two directions -- from the actual structure of the target database or from the questions that the targeted users would have for such a database. In fact, both directions are necessary to obtain a complete domain model for use in IRUS-II. An effective approach is to interview the targeted users and elicit samples of the actual questions they would submit to such a database, making sure each possible database table or field is covered by some question. This will define the range of concepts and relations people use in the domain.

It should be noted that the domain model is not required to be limited to contain entities that map to the database, but other related entities that a user is likely to use should be there too in order to respond sensibly when there are misconceptions by the user. For example, a target user of the database mentioned above might wish to ask a question such as

`In which factory was this helicopter made?`

IRUS-II should *understand* the question (be able to obtain a WML using domain predicates), even though it cannot answer it. In JANUS-B, a troubleshooting capability is under development to be able to answer "I don't have information about where helicopters are made".

If the domain developer is developing a domain model that is intended to be transportable to a number of different databases dedicated to the same domain, that domain model should contain all of the concepts that would possibly be reflected in those databases. IRUS-II will not support access to any data in a database unless that data is conceptually represented in the domain.

As an example of a domain, let us assume that our application is a hospital database. The targeted users are doctors, nurses, and other hospital staff. The hospital database will contain information about patients, doctors, nurses, floors, wards, rooms, etc. These will be the concepts in the domain model. One of the principles behind taxonomic networks is that it is useful to create concepts that generalize other concepts. This reduces the amount of information that has to be stored by taking advantage of inheritance. It also generates concepts that reflect semantic distinctions.

For example, it is likely that we would want to be able to query the database about a doctor's social security number. We would also like to be able to refer to the social security numbers of nurses and other staff members. Since it is likely that patient social security numbers are used for identification, we would also like to be able to refer to them. A sensible approach would be to create a concept called HOSPITAL-PERSON that is a superconcept of DOCTOR, NURSE, OTHER-STAFF and PATIENT. The Role SOCIAL-SECURITY-NUMBER could then be associated with the concept HOSPITAL-PERSON and inherited by all concepts subsumed by HOSPITAL-PERSON.

3.3 Building Domain Models with NIKL

Currently, IRUS-II domain models are built with NIKL [16]. NIKL (New Implementation of KL-ONE) is a system for representing conceptual knowledge. With NIKL, the user can represent knowledge in a structured inheritance network. In addition to enabling the user to describe a domain with constants that can be used in other modules of the IRUS-II system, NIKL also provides inference mechanisms that can be of use, such as inheritance and classification. A graphical interface, KREME [1, 2], is used to more easily visualize and modify a NIKL network.

In its current implementation, IRUS-II uses the inheritance mechanism. For example, the Semantic Interpretation Rules (IRules) test the semantic class of constituents to determine if they are subsumed by certain NIKL concepts. These tests can name the highest level concept for which they are appropriate, with confidence that the inheritance mechanism will correctly determine whether the individual being tested is a member of that class or any class subordinate to it in the network (subsumed by it).

Although it is not done in the current implementation, IRUS-II can also make use of the classification mechanism of NIKL. Classification enables NIKL to correctly place a new concept in a network from the definition of that concept. This makes it possible to take information in a query and use it to create a NIKL concept. This can be used to simplify an expression by reducing a complex expression to a NIKL concept. For further details, see [28].

3.3.1 NIKL Concepts

The objects in a NIKL domain are known as concepts. These concepts are organized in a network that is rooted in the concept THING, which *subsumes* all other concepts in the world. Concept A *subsumes* concept B if all instances of B (i.e., individuals of the world that are of class B) are instances of A. For example, all animals are living things. Therefore, if one were to build a network that included the concept ANIMAL, the concept HORSE, and the concept LIVING-THING, the concept LIVING-THING would subsume ANIMAL and HORSE. The concept ANIMAL would subsume the concept HORSE. Subsumption is transitive, i.e., if LIVING-THING subsumes ANIMAL, and ANIMAL subsumes HORSE, then LIVING-THING also subsumes HORSE. The relationship between a concept and the concepts that subsume it corresponds to an IS-A link in frame terminology.

Concepts can be either *primitive* or *defined*. Typically, domain development begins with the creation of some primitive concepts and then proceeds to the creation of defined concepts based on the existing primitive concepts. A primitive concept is one which the domain developer can not fully define in terms of existing concepts in the network.

A defined concept is a concept for which the necessary and sufficient criteria for membership in the class it represents can be expressed, by specifying the parents of the concept and appropriate role restrictions (the following section discusses role restrictions). For example, the necessary and sufficent criteria for membership in the class RED-SNEAKER is that an object be a SNEAKER and that it have the color RED, i.e., RED-SNEAKER is subsumed by SNEAKER and restricts the role COLOR-OF to have range RED. In many cases, the necessary and sufficient criteria for membership in the class represented by a concept can not be expressed. For example, consider the class TIGER. A TIGER is a subconcept of ANIMAL and we can express some of the necessary criteria for membership in the class represented by TIGER, but we can not define the necessary and sufficient criteria to distinguish a TIGER from all other members of the class ANIMAL. TIGER is thus a *primitive* concept, a concept that we can describe but not completely define.

3.3.2 NIKL Roles and Role Restrictions

Roles are NIKL entities which represent logical relationships between concepts. A Role would be referred to as a *slot* in frame terminology. A Role is a two-place relation whose domain is represented by a concept and whose range is represented by a concept. A Role maps each instance of the domain concept into instances of the range concept. An instance of the range concept is a *filler* of the role for the instance of the domain concept.

Part of the description of a concept may include role restrictions. Role restrictions restrict the allowed class of the range and/or the number of range fillers for that role, when the domain is that concept. Subconcepts inherit both the Roles and RoleRestrictions of their superconcepts. Inherited RoleRestrictions can be further restricted, but not generalized, for the subconcept. In fact, this is one way of defining a subconcept. For example, the concept PERSON may have the Role ADDRESS attached to it. The range of ADDRESS may be restricted to addresses. The concept USA-RESIDENT may be *completely* defined as the subconcept of PERSON for which the range of ADDRESS is restricted to addresses in the USA.

The NUMBER component of a role restriction is particularly important when defining a *functional* role, i.e., a role that relates each element of the domain to one and only one instance of the range. HULL-NUMBER-OF is one such role relating VESSEL to HULL-NUMBER. To define a functional role, one must have a role restriction with the NUMBER component specified as 1 at the most general concept that is the domain of the role. IRUS-II does not use this information currently, but will in the future when a more general treatment of functional nouns is implemented.

When one is developing a NIKL network, one goal is to give roles the most general domain and range that are relevant, and when a role restriction is needed, to connect it to the most general concept possible.

3.4 NIKL Syntax

The two primary functions used in creating a domain model with NIKL are *defconcept* and *defrole*. As the names indicate, defconcept is used to create a concept and defrole is used to create a role. The following uses standard BNF syntax, where { } denotes optional items, and * denotes one or more repetitions of an expression.

3.4.1 DEFCONCEPT

The format for defconcept is:

```
(DEFCONCEPT <concept-name>
            (SPECIALIZES <concept>*)
            {PRIMITIVE}
            {INDIVIDUAL}
            {(RES <role> {(VRCONCEPT <concept>)
                         {(NUMBER <integer>)}
                         {(MIN <integer>)} {(MAX <integer>)})}*
            {(DATA (LEXICALITEMS (<lexical-item>)))}

        )
```

PRIMITIVE

means that the necessary and sufficient criteria for membership in this class are not expressed in the domain model. If not present, assumed to be a DEFINED concept.

INDIVIDUAL

means that the concept being created represents an individual entity which is a member of the class or classes denoted by this concept's parents. For example, the current domain model includes the following:

```
(DEFCONCEPT TRUE PRIMITIVE INDIVIDUAL (SPECIALIZES TRUTH.VALUE))
```

TRUE is a an individual of class TRUTH.VALUE (the only other member of the TRUTH.VALUE class is the individual FALSE).

SPECIALIZES

specifies the concepts that immediately subsume the concept being created.

RES

expresses restrictions on a role (which is named immediately after the keyword), when the domain of the role is an instatiation of this concept. These restrictions are specified with VRCONCEPT, NUMBER, MIN, and MAX.

VRCONCEPT

specifies a class restriction on the filler of the range of role. The filler of the range is limited to members of the named class.

NUMBER

specifies the exact number of individuals that may fill that role for each member of the class. For example, the current domain model includes the following:

```
(DEFCONCEPT VESSEL
            PRIMITIVE
            (SPECIALIZES PLATFORM)
            (RES COMMANDER.OF (VRCONCEPT COMMANDER) (NUMBER 1)))
```

The last line indicates that the role COMMANDER.OF associated with the concept VESSEL may be filled by exactly one member of the class COMMANDER.

MIN

Similar to NUMBER, but specifies a minimum number of role fillers for each member of the class.

MAX

Similar to NUMBER, but specifies a maximum number of role fillers for each member of the clas.

DATA

is rarely used. In conjunction with LEXICALITEMS it specifies an English word associated with the concept. Interpretation Rules (discussed in Chapter 4) are connected to domain objects via this data field, but this is not specified in the domain model file, they are added by the IRule loader program.

3.4.2 DEFROLE

The format for defrole is:

```
(DEFROLE <role-name>
         (DIFFERENTIATES <role-name>*)
         (DOMAIN <concept>)
         (RANGE <concept>)
         (PRIMITIVE)

        )
```

DIFFERENTIATES

specifies a parent role which the role being created is a subrole of (a more specific relation), that is, if the parent role is viewed as denoting a set of ordered pairs, this role's denotation is a subset of that set. The role being created must have as domain and range concepts that are subconcepts of the parents' respective domains and ranges.

DOMAIN

specifies the domain of the role.

RANGE

specifies the range of the role. The role represents a relation between elements of the domain and elements of the range.

PRIMITIVE

A primitive role is one for which necessary and sufficient criteria can not be expressed. If not present, the role is assumed to be a DEFINED role.

3.5 KREME

BBN's Knowledge Representation, Editing, and Modeling Environment (KREME) is an excellent tool for the development of a domain model. KREME has its own frame language which is closely related to NIKL, but KREME also has a NIKL mode which enables it to work with domain models built with NIKL. Whether the domain model is created with NIKL directly or via KREME in the NIKL mode, the result is still a NIKL taxonomy. The only difference is that KREME provides a better environment for domain model development. For further information on KREME, consult *KREME: A User's Introduction*, BBN Report No. 6508.

3.6 A Brief Domain Model Example

This section is intended to provide some insight into the process involved in developing a domain model for a specific application. Let us assume that we wish to construct a domain model for a naval application. In this application a number of different types of objects will be referred to. Among them are:

- helicopter
- airplane
- submarine
- carrier
- ocean

- port
- weapon

Each of these objects will be represented as concepts. If we did nothing more than that, we could create a network whose top level was THING and whose next level would represent each of these objects as concepts. However, we would not have succeeded in capturing any generalizations about these concepts and we would not have established any inheritance (beyond the fact that they all descended from THING) which we could later use in our IRules. Therefore, we would wish to create some additional concepts which expressed generalizations about these objects.

One approach would be to sort them into groups and then to create concepts that represent those groups. For example, one might create a group from helicopter, airplane, submarine, and carrier, and call that concept VEHICLE. Intuitively, one feels that the distinction between air vehicles and sea vehicles might be a useful one and so might create the concepts AIRCRAFT and VESSEL. These would each be subsumed by VEHICLE. In turn, AIRCRAFT would subsume the concepts HELICOPTER and AIRPLANE, and VESSEL would subsume the concepts SUBMARINE and CARRIER.

The concepts OCEAN and PORT could be subsumed under a concept we will call LOCATION.

The concepts READINESS and WEAPON can be subsumed by THING.

These concepts will suggest roles that are needed to express their attributes. These roles in turn will require the creation of additional concepts to represent their ranges. For example, we may wish to describe the location of a vehicle. We could create the role VEHICLE.LOCATION with the domain VEHICLE and the range LOCATION. We might also wish to refer to the commander of a VESSEL. We would create the role COMMANDER.OF and have its range as OFFICER. It is likely that before we finish we will want to represent other people, so we might subsume OFFICER under the concept PERSON. All of these domain model decisions should be considered tentative until we have explored the domain further.

One way to futher explore the domain is to get a list of sentences which would typically be input by the user. Let us assume that the following sentences appear in this list:

What is the overall readiness of the Enterprise?

What is the Enterprise's home port?

[Enterprise is the name of a carrier]

These suggest some additional concepts and roles. The existing concept VESSEL will be the domain of the role OVERALL READINESS. This will have a new concept, perhaps READINESS.CODE, as its range. The role HOME.PORT might be created with the concept VESSEL as its domain, and PORT as its range.

Again, these domain model decisions should be regarded as preliminary until it is established that the underlying system contains the information necessary to respond to such requests.

The domain developer would proceed to develop a domain model, using anticipated user input and the information contained in the underlying system as guides. The concepts and roles in this preliminary model and the anticipated input sentences would then be used to develop the IRules and the dictionary entries. This process would provide feedback on the appropriateness of the domain model. The domain developer should be prepared to make changes to the domain model in response to this feedback.

4. Semantic Interpretation Rules

A Semantic Interpretation Rule (IRule) consists of two sections, a pattern section and an action section. The pattern section is a case frame. Constituents of a query may fill slots in a case frame if they meet the selection restrictions attached to that slot. The action section contributes to the logical representation of the query. The logical representation of the query is in the World Model Language (WML), described in chapter 6.

The Semantic Interpretation Rules (IRules) are linked to particular domain model concepts or roles when they are defined by the domain developer. They are then activated by words in the input query, whose dictionary entries link them to corresponding domain concepts. The IRules define, for a particular word or class of words, the semantically acceptable English phrases or clauses that can occur having that word as their head. They also define the semantic interpretation of an accepted phrase or clause. Since semantic processing is integrated with syntactic processing in IRUS, the IRules serve to block a semantically unlikely interpretation as soon as it is proposed by the parser.

The parser passes ("transmits") constituents to the semantic interpreter. The semantic interpreter attempts to match each constituent to the pattern sections of the IRules associated with the head word of the current phrase. If the pattern section of an IRule is successfully matched, that IRule's action section is queued for execution. The action sections produce the fragments of WML that are ultimately composed into the system's logical representation of the query. The action sections' secondary function is to place additional information on a list of features that accompany the partial interpretations and which other IRules may access, such as the semantic class of the constituent. Failure to match also provides useful information to the parser. If a particular parse route fails to find a corresponding semantic interpretation, then the parser tries another parse route.

When a head word of a phrase is proposed by the parser to the semantic interpreter, all IRules that can apply to the head word for the given phrase type are gathered as follows: for each domain model object (concept or role) that is associated with the word, the IRules associated with the given object are retrieved, along with any inherited IRules. A word can also have IRules fired directly by it, without involving the model. Since the IRules corresponding to the different word senses may give rise to separate interpretations, they are carried along in parallel as the processing continues. If no IRules are retrieved, the interpreter rejects the word.

One use of the domain model is that of IRule inheritance. When an IRule is defined, it is attached to a concept in the domain model. The definer of the IRule also has the opportunity to decide whether the new IRule (the *base* IRule) should inherit from IRules attached to higher domain model concepts (the *inherited* IRules), or possibly inherit from other IRules specified by the definer of the IRule. When a modifier of a head word gets transmitted and no pattern for it exists in a base IRule for the head word, higher IRules are searched for the pattern. If a pattern does exist for the modifier in a given IRule, no higher ones are tried even if it does not pass the local semantic test. That is, inheritance does not relax semantic constraints.

Generally speaking, IRules are attached to concepts or roles in the domain model and they are indirectly linked to words in the lexicon by the inclusion of the names of domain model objects in the semantics of each dictionary entry. When no domain object seems appropriate as a semantic entry for a word, IRules may be linked to the word directly by having the name of the IRule included in the semantics of the word.

4.1 Pattern Section

Once the head of a phrase triggers an IRule, all of its modifiers must match some slot in the pattern section of the IRule. If the matches succeed, the action section of that IRule is queued for execution at the time the constituent is completed ("constituent pop time").

There are two types of IRules, CLAUSE IRules and NP IRules. CLAUSE IRules are further divided into VERB, PP, and ADJ IRules. (It is planned that PP and ADJ IRules will in the future be separate categories altogether, so that there will be 4 general categories of IRules.) Most simple cases of NP, PP, and ADJ IRules may now be defined using KNACQ (chapter 7).

4.1.1 Clause IRules

The following is a template for the pattern section of a VERB clause IRule. Its head is a verb, and the IRule defines the case frame for that verb. Standard BNF syntax is used.

```
(CLAUSE
        HEAD            *
        {SUBJECT        <test>}
        {OBJECT         <test>}
        {INDOBJ         <test>}
        {PP             ((PP HEAD       <Preposition-test>
                             POBJ       <test>)+)}
        {THATCOMP       <test>}
        {<other-verb-modifier-label> <test>}*
        )
```

CLAUSE means that this IRule is designed to handle clause-level constituents, either sentences or subordinate (e.g. relative) clauses. NP IRules would handle the NP subconstituents of the sentence. It is helpful to think of the

clause IRules as the higher level IRules that bind together subinterpretations produced by the NP IRules. Following "CLAUSE" is a property list, of <parser-label> and <test> pairs, each pair representing a selectional restriction. Every structure "transmitted" by the parser to the interpreter has a parser-label; the test places semantic constraints on the structure transmitted with the given label.

HEAD is a slot that must be filled by the head of the constituent. The HEAD of a (non predicate-adjective) sentence is its verb, the head of an NP is a noun, the head of a sentence with a predicate adjective ("<NP> <be> <adjective>") is the adjective (e.g., the parser considers "pretty" as the head in "the girl is pretty"), and the head of a sentence with a predicate PP is the copular "BE" (e.g., "the ship IS in Hawaii"). Since the semantic properties attached to a specific verb trigger clause IRules during the interpretation of a query, there are typically no additional restrictions on the HEAD. This is represented by the asterisk associated with the HEAD slot.

The *SUBJECT* of a clause is usually a noun phrase. In order to be accepted as a filler for the SUBJECT slot, a noun phrase must meet the test associated with that slot. Section 4.1.5 presents the syntax of these tests.

The *OBJECT* of a clause is the direct object of the (transitive) verb.

The *INDOBJ* is the indirect object. For example, the verb *show* may take an indirect object:

```
Show me the ships in the Indian Ocean.
```

The direct object of show is *ships* and the indirect object is *me*.

Verbs may be modified by prepositional phrases. The *PP* slot of a clause IRule would be filled by such prepositional phrases. This slot may not be filled by a prepositional phrase that modifies a noun. These are handled in NP IRules or PP (predicate prepositional phrase) IRules. The HEAD of a prepositional phrase is the preposition and the POBJ is the object of the preposition. There are two tests on prepositional phrases; one on the head preposition and another on the object of the preposition.

The PP slot may be repeated because more than one PP may modify a clause. Repeated slots are implicitly assigned numbers that are later used in the action section. For example, the current IRule *CHANGE.IN.READINESS.2* (triggered by the verb *change*) has these PP slots associated with it:

```
(PP     HEAD    FROM    POBJ (IS-A C.READINESS.CODE))

(PP     HEAD    TO      POBJ (IS-A C.READINESS.CODE))
```

In the action section, these are distinguished from each other by references to PP 1 and PP 2. Different WML representations are built for each case.

THATCOMP tests for a THAT complement, as in "Frederick reported that Vinson downgraded".

There may be selectional restrictions for any other clause modifier such as TOCOMP (TO complements) that the parser may transmit.

4.1.2 NP IRules

The format for the pattern section of an NP IRule looks like this:

```
(NP
        HEAD            *
        (NP             (<test>+) )
        (PP             ((PP HEAD <Preposition-test>
                             POBJ <test>)+) )
        (N              (<test>+) )
        (ADJ            (<test>+) )
        (PRESPART       (<test>+) )
        (PASTPART       (<test>+) )
        (PRENOMINAL     (<test>+) )
        )
```

NP means the IRule is designed to handle parts of noun phrases. A noun phrase includes a noun plus all its modifiers such as articles, adjectives, noun modifiers, and prepositional phrases.

The *HEAD* of a noun phrase is the noun itself. Since the noun would typically have the appropriate semantic property to trigger the IRule, usually there are no restrictions on the HEAD. This is expressed with the asterisk. It is possible, however, to put a test on the head of an NP IRule.

NP is for noun phrase modifiers to the head noun. NP, like the other selectional restrictions presented so far (except for HEAD), tests full (interpreted) constituents. *Interpreted* constituents have been interpreted as phrases and have already triggered IRules of their own. Interpreted constituents already have a partial WML and semantic class associated with them. For example, an NP selectional restriction could accept an NP constituent for "USA" modifying "ships" in the noun phrase "USA ships", as well as accepting "friendly country" in "friendly country ships". An algorithm for noun compounds in IRUS-II tries to break up a sequence of nouns into component NPs.

Other modifiers of nouns, whether prepositional phrases *PP*, adjectives *ADJ*, past participles *PASTPART* (e.g., "the *crossed* river") or present participles *PRESPART* (e.g., "the *deploying* commander"), are normally handled by PP rules, adjective rules or verb clause rules, respectively. The only reasons to put tests for these modifiers directly in an NP IRule are (1) to handle idiomatic readings ("the *red* (Russian) forces"), or (2) because the modifier never occurs in predicate position, like "of the ship" in "captain of the ship" but not (*)"captain is of the ship". Only if the meaning of a modifier is different when it is in a noun phrase from when it is a predicate modifier should a test for this modifier be put directly in the NP rule. Otherwise it need not be mentioned in the NP IRule. For example, the UNIT.BE.PREDICATE.1 PP IRule handles both "List the ships in port" and "List the ships that are in port".

The *PP* is as for CLAUSE Irules.

It should be noted that *N*, *ADJ*, *PRESPART*, *PASTPART*, and *PRENOMINAL* test for *uninterpreted* constituents. An uninterpreted constituent does not carry a semantic interpretation with it. Thus if an uninterpreted constituent is bound to a variable in an IRule and used in constructing a WML predicate, the lexical head itself is what will appear in the WML in place of the IRule variable. In the case of *interpreted* constituents, however, a term--a IOTA expression or a variable bound by an outer scoping expression which is the interpretation of the transmitted constituent--is what is replaced for the IRule variable. PRENOMINAL is a super-label; it accepts Ns, ADJs, PRESPARTs, and PASTPARTs.

4.1.3 Adjective IRules

Adjective clause IRules are triggered when an adjective is used in predicate position (e.g., "the girl is *pretty*", "the girl that is *pretty*"), where the adjective (e.g., *pretty*) is considered the head of the clause. Also when an adjective appears directly modifying a noun (e.g., "the *pretty* girl"), the semantic interpreter tries to use the general meaning of the adjective encoded in the appropriate adjective rule to obtain an interpretation, by interpreting it as a predicate adjective in a relative clause ("the girl that is *pretty*"). (In this NP case, the interpreter will first try an idiomatic interpretation by checking the head's NP IRule for an ADJ test, and will keep this result as a first option in the list of possible NP interpretations.)

The following is a template for the pattern section of an adjective clause IRule:

```
(CLAUSE
        HEAD            *
        SUBJECT         <test>
        (PP             ((PP HEAD       <Preposition-test>
                             POBJ       <test>)+))
        (THATCOMP       <test>)
        (<other-verb-modifier-label> <test>)*
        )
```

HEAD is the adjective, which is considered head of the clause when it appears in predicate position.

SUBJECT tests the NP being modified by the adjective.

PP and other labels are as for verb clause IRules. They are present when the ADJ takes complements. For example, "afraid" takes a THATCOMP in "I am afraid that I will fail" and mission area ratings take PP complements, as in "Frederick is M1 on ASW".

4.1.4 Prepositional Phrase IRules

Prepositional phrase clause IRules are triggered when a PP is used in predicate position, e.g., "the ship is *in Hawaii*", "the ship that is *in Hawaii*". An attempt is made to handle PPs analogously to ADJs, since they both occur in predicate position as well as directly in noun phrases modifying the head NP, but both syntactic positions yield the same meaning. Thus, as in the ADJ case, when a PP appears directly modifying a noun (e.g., "the ship *in Hawaii*"), the semantic interpreter tries to use the appropriate PP rule to obtain an interpretation by interpreting it as a predicate PP in a relative clause after first trying an idiomatic interpretation. If the KNACQ system is in use, the interpretation of a direct PP modifier as a relative clause PP does not happen with KNACQ, separate PP rules need not be defined, it handles the PPs automatically when an attribute or case rule is defined.

A difference from the ADJ treatment is that the parser does not consider the PP to be the head of the phrase when in predicate position, but insists to have "BE" as the head. Therefore PP IRules must be triggered by "BE". Currently, PP IRules must be all inherited by the default IRule NP.BE.PREDICATE. We plan to change the way PPs are treated to be more analogous to ADJs.

Following is a schema for the pattern portion of PP clause IRules:

```
(CLAUSE
        HEAD            BE
        SUBJECT         <test>
        PREDICATE       ((PP HEAD       <Preposition-test>
                             POBJ       <test>)+)
        )
```

HEAD must be "BE".

SUBJECT tests the NP being modified by the prepositional phrase.

PREDICATE has the syntax of PP in the other IRule types. It tests for the PPs that should trigger this IRule. More than one PP may be listed in the same IRule, as long as the selectional restriction on what they modify (the SUBJECT) is the same.

4.1.5 Selectional Restriction Tests

A test associated with a label in an active IRule is applied to any constituent transmitted with that label. If the constituent fails the test, that particular parse path is aborted by the parser, and another path is tried. If there is no test for the transmitted label, inherited IRules will be tried to accommodate the transmitted constituent. If no inherited IRule accepts it, it is rejected. When a transmitted constituent is accepted, it is saved to be used by the action portion of the IRule when the current phrase is finished.

An IRule label may be a *unique item label* if only one constituent may fill that slot (e.g., SUBJECT), or it may be a *multiple item label* if it can accept more than one filler (e.g., PP, PRENOMINAL). When the label is a unique item label, it is associated with a single test in the IRule, else a list of tests is used, indicating the different possible fillers.

A test associated with an IRule label is either a *base* test, or a path specification containing base tests in it. For example,

```
(PP HEAD (ONE-OF OF FROM) POBJ (IS-A COUNTRY)))
```

is a path specification with two base tests: (ONE-OF OF FROM) and (IS-A COUNTRY), testing for the PP head and its POBJ, respectively. Using general path specifications, except for the usual PP schema as above, is not recommended.

There are several types of base tests that can be associated with labels in an IRule.

- **Simple semantic test**: Normally of the form (IS-A <domain-concept-name>). A constituent passes this test if its semantic class is subsumed by or it subsumes the concept denoted by <domain-concept-name>. This IS-A is really a two-way subsumption test in order to allow more general concepts to fill the semantic constraint, as is necessary for pronouns, for example. In "He deployed the ship", "he" has semantic class "PERSON", a superconcept of "COMMANDER", which the verb "deployment" requires as subject. IS-A-SUPERC instead of IS-A may be used if strict one-way subsumption is required.
- **Exact match**: A test for an exact match with a constituent may be done in one of three ways:
  - (EXACTLY <word>) will test whether the root of the head of the constituent is <word>.
  - <word> will test whether the *uninterpreted* constituent is equal to <word>. Used to test the preposition head of a PP, for example.
  - (ONE-OF <word1> <word2>...<wordn>) is like <word>, but is used when more than one option is acceptable.
- **Arbitrary test**: of the form (TEST <arbitrary-test>). <Arbitrary test> may be a simple boolean combination (using NOT, AND, OR) of simple IS-A tests, or may be a function call to an arbitrary function. Use of arbitrary functions is not recommended.
- **Automatic success**: if the constituent should be accepted regardless, T or (TEST T) may be used. Another test which immediately succeeds is *, but this should only be used for IRule heads, because constituents approved via * cannot be replaced by pronouns.
- **Forced failure and block**: to cause immediate failure when a constituent is present, and prevent inherited rules to be tried after the failure, (TEST NIL) may be used.

4.2 Action Section

The primary function of the action section of an IRule is to contribute to the production of the WML that will represent the phrase which triggered the IRule. Another function is to specify any modifiers that may be required (not optional) for the IRule to fire.

The entire action section of an IRule is a macro call. In a clause IRule, it is a call to the macro BINDQUANTS; in an NP IRule it is a call to the macro LIFTQUANTS. These macro names are meant to be mnemonic: quantified expressions with possible "holes" in them (to be filled by other parts of the sentence) stemming from NPs are carried ("lifted") until a clause boundary is reached, at which point they are "bound" together with the clause's contribution, i.e., all the holes are filled and the quantification structure of the clause interpretation is determined. These macros take as the first argument a variable binding list. The other arguments (the body of the action) are actions to be performed using the bound variables.

4.2.1 Variable Binding List

The first argument to LIFTQUANTS and BINDQUANTS in the action part of an IRule is a variable binding list. Variable names are bound to fillers of constituent labels mentioned in the left hand side of the IRule.

The syntax of the bindings list is

```
((<var1> <constituent-description1>)
 (<var2> <constituent-description2>)
      ...
 (<varn> <constituent-descriptionn>))
```

Where the syntax of a <constituent-description> may be given by the following context free definition, where | signifies "or", { } is for optional items, and a superscript of + means one or more repetitions:

```
constituent-description -> c1 | (OPTIONAL c1) | (DEFAULT c1 <default>)
      | (LISTOF C1)
c1 -> (OR c2+) | c2 | c3
c2 -> c3 | (PROPERTY <c3> <semantic-prop>) | (FULL-INTERP <c3>)
c3 -> (or <path>+) | path
```

OPTIONAL means that even if the specified label has not been filled, the IRule execution is still allowed to complete. The variable is assigned value **ABSENT** and any action in the body using the variable is ignored.

DEFAULT allows for a default value <default> to be used as the variable value if the specified label is not filled. OPTIONAL and DEFAULT allow for a label filler to be missing. If neither of these is used, the filler is considered *required*. If a required filler is not present, the IRule execution fails, i.e., an interpretation using this IRule is no longer considered. (If it was the only IRule, the constituent "pop"--finishing the phrase--will fail, and the parser will have to try another parse path.) LISTOF obtains a list of all possible fillers of the description.

OR states that any of the specified labels may be used to bind the variable. The first label from the list that has a filler is used (except in the context of a LISTOF, where all are used).

<path>, in its simplest form, is just a label, e.g. SUBJECT. It may be a more complex path, as is the case with multiple fillers for a label such as PRENOMINAL. In this case a path would be (PRENOMINAL <n>), where <n> is an integer, indicating the nth PRENOMINAL test from the left hand side of the IRule. Complex paths for PPs are typical, e.g. (PP 1 POBJ) meaning the object of the first PP.

PROPERTY takes a label or path and a domain model object name (a concept or role). The semantics field (in the lexicon) for the head of the label filler is searched, and if the object name is in the semantics, the variable is bound to that semantic property's value (usually *, or something complex like (NAMEOF "FREDERICK") or (CODE 6A)). A variant of this is using (ANY <semantic-prop>) instead of <semantic-prop>. When ANY is used, any superconcept (or superrole) of <semantic-prop> (including <semantic-prop> itself) is used if it is found in the head's semantics.

FULL-INTERP is used when one wants to bind the variable to the whole WML interpretation of the label filler. FULL-INTERP should only be used for intensional arguments. For example the CHANGE IRule uses (FULL-INTERP SUBJECT) to bind an IRule variable to the subject, in order to apply the INTENSION operator to it in its interpretation. Normally the IRule variables get bound to a IOTA term (in the definite NP case), to a SETOF expression (in the case of an NP conjunction), or to a variable that is bound by the scoping WML expression generated for the label filler. FULL-INTERP causes all partial WML pieces that were being "lifted" to be "bound" into a closed expression. This closed expression is made the value of the IRule variable.

4.2.2 IRule Body

The body of the action part of the IRule is composed of individual actions (function calls) that add bits of WML to the interpretation of the current phrase that is being formed. Normally one scoping expression is generated per NP or clause. The primary function of the actions in the body is to specify the *sort* (semantic class) of the variable to be used in the scoping expression and to specify predicates that will be gathered for the scoping expression's body.

Typical actions that may be found in IRules:

(ANDPREDICATE '(<rolename> <var1> <var2>)) : Adds the given predication to the conjuncts that are being gathered for the expression's body. *V* is used to refer to the local variable of the scoping expression under construction. ANDPREDICATE is used in NP IRules.

(SEM '(<rolename> <var1> <var2>)) : analogous to ANDPREDICATE. SEM is used for CLAUSE IRules.

(CLASS <conceptname>) : specifies the semantic sort to be used in the quantified expression.

(LOCALQUANT '<det>) : Where <det> may be SOME or THE, which makes the branch category of the scoping expression be EXISTS or IOTA, respectively. For NP rules, LOCALQUANT is not necessary, since the branch category is determined by the determiner used in the noun phrase. For CLAUSE IRules, this should be SOME if a quantified expression is desired for the CLAUSE. If only predications are desired, SEMs may be used alone, or (NOLOCALQUANT) may be used. (We will probably remove the need to use LOCALQUANT at all soon.)

(IF-BOUND
(<var1> <action11> <action12>...)
...
(<varn> <actionn1> <actionn2>...)
(otherwise <action1> <action2>...))

IF-BOUND is analogous to a LISP CASE statement. It executes the actions in the first branch that names a bound IRule variable.

The following are two utility functions that may be used as part of an IRule action:

(GET-IS-A-SEMPROP '<word> '<domain objectname>) : This function searches the semantics field of <word> in the lexicon, trying to find a semantic property (a "sense") of the word which names a domain object equal to or subsuming the object named by <objectname>. It returns the word sense that matched. GET-IS-A-SEMPROP is usually used to find the most specific semantic property of <word>, which is known to be a subconcept or subrole of <domain object>.

(MAKE-IRULE-SCOPING-EXPRESSION '<predication1> ... '<predicationn>
:class <sort> :branch-cat <branch-category>) :
If more than one quantified expression is absolutely required to be generated for one IRule, MAKE-IRULE-SCOPING-EXPRESSION creates one that may be used as an argument to an ANDPREDICATE or SEM, or as part of another predication. <Predication> may be in terms of a special variable *E* which refers to the local variable to be created for that expression.

4.2.3 IRule Examples

This section provides an analysis of two IRules: one is an example of an NP IRule and the other a clause IRule.

4.2.3.1 An NP IRule

```
(IRule
 (IRule BATTLE.GROUP.1
   (NP HEAD *
       PP ((PP HEAD OF POBJ (IS-A UNIT))
           (PP HEAD IN POBJ (IS-A LOCATION.IN.SPACE))))
   ==>
   (LIFTQUANTS ((UNIT.1 (OPTIONAL (PP 1 POBJ)))
                (PLACE.1 (OPTIONAL (PP 2 POBJ))))
               (ANDPREDICATE '(IN.PLACE *V* PLACE.1))
               (ANDPREDICATE '(UNIT.GROUP UNIT.1 *V*))
               (CLASS BATTLE.GROUP)

 SUPERRULE T CONCEPT-IRule-ATTACHED-TO BATTLE.GROUP)))
```

The head of a noun phrase handled by this IRule can be modified by either or both of two types of prepositional phrase. The pattern for PP 1 accepts a prepositional phrase with the preposition "of" and an object that is a "UNIT". PP 2 accepts a prepositional phrase whose preposition is "in" and whose object is a "LOCATION.IN.SPACE". UNIT and LOCATION.IN.SPACE are concepts from the domain model.

If a constituent matches this pattern, then the action section will be executed. The action section binds the optional object of PP 1 to the variable UNIT.1. It also binds the optional object of PP 2 to the variable PLACE.1.

Two ANDPREDICATE functions then add predications to be used in the body of the scoping expression to be created. *V* is a variable which is automatically bound to the local variable that is created for this expression.

(CLASS BATTLE.GROUP) sets the sort field of the expression, and labels this constituent as having class BATTLE.GROUP.

If the input NP is "the battle group in the Indian Ocean", the determiner "the" would result in a IOTA branch category, and the partial WML interpretation would be:

```
(IOTA ?JX1 BATTLE.GROUP
    (IN.PLACE ?JX1 <"Indian Ocean" interp>))
```

Since there was no "OF" PP, the UNIT.1 IRule variable was not bound and the ANDPREDICATE using it was ignored.

4.2.3.2 A Clause IRule

```
(IRule
 (IRule DEPLOYMENT
  (CLAUSE SUBJECT (IS-A COMMANDING.OFFICER) HEAD * OBJECT (IS-A UNIT)
     PP ((PP HEAD TO POBJ (IS-A LOCATION.IN.SPACE))))
  ==>
  (BINDQUANTS ((CO.1 (OPTIONAL SUBJECT)) (UNIT.1 (OPTIONAL OBJECT))
               (TO.1 (OPTIONAL (PP 1 POBJ))))
              (LOCALQUANT 'SOME)
              (CLASS DEPLOYMENT)
              (SEM '(AGENT *V* CO.1))
              (SEM '(OBJECT *V* UNIT.1))
              (SEM '(DEPLOYED.TO *V* TO.1)))))
```

The head of the clause handled by this IRule must have a subject which is a "COMMANDING.OFFICER". If it takes a direct object, it must be an object which is a "UNIT". The verb may also be modified by a prepositional phrase. If it is, the preposition must be "to" and the object of the preposition must be a "LOCATION.IN.SPACE" COMMANDING.OFFICER, UNIT, and LOCATION.IN.SPACE are concepts from the domain model.

If a constituent matches this pattern, then the action section is executed. Because this is a clause IRule, the action section is a BINDQUANTS macro. The action section binds the subject to the variable ACTIVE.ENTITY.1. It also binds the direct object to the variable UNIT.1 and the object of the preposition to the variable TO.1.

If the input was "An admiral deployed Frederick to the Indian Ocean", the piece of WML built by this IRule would be:

```
(EXISTS ?JX1 DEPLOYMENT
   (AGENT ?JX1 <"an admiral" interp>)
   (OBJECT ?JX1 <"Frederick" interp>)
   (DEPLOYED.TO ?JX1 <"Indian Ocean" interp">))
```

4.3 Creating IRules

Developers who are building IRules for a new domain or extending the IRules in a given domain may use the IRules Acquisition interface (IRACQ) (best suited for defining verbs), KNACQ (chapter 7), or write the rules directly into an editor.

Creating IRules involves constructing a case frame pattern section for a query constituent and a template to build a WML representation of the query (action section). Since the WML representation of the query uses terms from the domain model, the IRule developer must understand the domain model for a particular domain before writing IRules for that domain.

4.3.1 Using IRACQ

IRACQ is a tool that a developer can use to add new IRules to a system. An alternative to IRACQ is to directly enter and edit an IRule in an editor. The advantage of IRACQ is that it prompts the developer and leads him or her through an interactive session that makes the creation of new IRules a relatively easy task. IRules are defined by the user using sample phrases: sample noun phrases for NP IRules, sample sentences for verb clause IRules (with the verb as head), sample sentences with a predicate adjective (with adjective as head) for ADJ clause IRules, and sample sentences with predicate PPs (with "BE" as head) for PP clause IRules. It is possible that the developer may wish to create a complex IRule that would not be supported via IRACQ. Even when the developer chooses to use IRACQ, the option of editing the resulting IRule is provided.

4.3.2 An IRACQ session

In this section we step through the definition of a clause IRule for the word "send". We assume that lexical information about "send" has already been entered.

The user enters the word that he wishes to create an IRule for, the name of the domain model concept he wishes to connect the IRule to, and a sentence using the new word. From the example presented by the user, IRACQ must learn which unary and binary predicates are to be used to obtain a representation. Furthermore, IRACQ must acquire the most general semantic class to which the fillers of the slots must belong.

Output from the system is shown in bold face, input from the user in regular face, and comments are inserted *in parentheses*

Word that should trigger this IRule: send

Domain model term to connect IRule to deployment

Enter an example sentence using "send":
An admiral sent Enterprise to the Indian Ocean

Choose a generalization for *COMMANDING.OFFICER*

COMMANDING.OFFICER PERSON CONSCIOUS.BEING ACTIVE.ENTITY OBJECT THING

(The user's selection specifies the case frame constraint on the logical subject of 'send'. The user picks COMMANDING.OFFICER. IRACQ will perform similar inferences and present a menu for the other cases in the example phrase as well, asking each time whether the modifier is required or optional. Assume that the user selects UNIT as the logical object and REGION as the object of the preposition "to".)

Which of the following predicates should relate *DEPLOYMENT* to *REGION* in the *WML*?:

LOCATION.OF DESTINATION.OF

(IRACQ presents a menu of binary predicates relating DEPLOYMENT and COMMANDING.OFFICER, and one relating DEPLOYMENT and UNIT. The user picks AGENT and OBJECT, respectively.)
Enter examples using "send" or <CR> if done:

(The user may provide more examples. Redundant information would be recognized automatically.)

Should this IRule inherit from higher IRules? yes

This is the IRule you just defined:

```
(IRule DEPLOYMENT.4
 (clause subject (is-a COMMANDING.OFFICER)
      head *
      object (is-a UNIT)
      pp ((pp head to pobj (is-a REGION))))
 ==>
 (bindquants ((commanding.officer.1 (optional subject))
         (unit.1 (optional object))
         (region.1 (optional (pp 1 pobj))))
   (andpredicate '(destination.of *v* region.1))
   (andpredicate '(object.of *v* unit.1))
   (andpredicate '(agent *v* commanding.officer.1))
   (localquant 'SOME)
   (class 'DEPLOYMENT)))
```

Do you wish to edit the IRule? no

(The person may, for example, want to insert something in the action part of the IRule that was not covered by the IRACQ questions.)

The sense of "sending" we have defined specifies an event type whose sorted lambda representation is as follows:

$(\lambda x : deployment)$
$agent(x, a) \ \& \ object(x, o) \ \&$
$\quad destination(x, d))$

where the agent a must be a commanding officer, the object o must be a unit and the destination d must be a region.

This concludes our sample IRACQ session.

4.3.3 Debugging environment

The facility for creating and extending IRules is integrated with the IRUS NLI itself, so that debugging can commence as soon as an addition is made using IRACQ. The debugging facility allows one to request IRUS to process any input sentence in one of several modes: asking the underlying system to fulfill the user request, generating code for the underlying system, generating the semantic representation only, or parsing without the use of semantics (on the chance that a grammatical or lexical bug prevents the input from being parsed). Intermediate stages of the translation are automatically stored for later inspection, editing, or reuse.

IRACQ is also integrated with the other acquisition facilities available. As the example session above illustrates, IRACQ is integrated with KREME, a knowledge representation editing environment. Additionally, the IRACQ user can access a dictionary package for acquiring and maintaining both lexical and morphological information.

Such a thoroughly integrated set of tools has proven not only pleasant but also highly productive.

4.3.4 Editing an IRule

If the user later wants to make changes to an IRule, he/she may directly edit it. Currently, there are no checks performed on the edited IRule to ensure that it is consistent with the domain model.

5. The IRUS Dictionary

5.1 Introduction

The dictionary interface allows a user to create, edit, and save dictionary definitions for the IRUS system without having to know the details of how the definitions are represented. However, the user must have some understanding of the category and feature system of IRUS.

Categories describe the general syntactic behavior of a particular lexical item. In addition, the categories *ADJ*, *N*, *PRO*, *POSSPRO*, and *V* may have a value that describes the way the lexical item is inflected.

The following dictionary entries provide some idea of the inflectional information (and other information) that different categories can have:

Verbs

```
[BLOW   FEATURES  (INDOBJ INTRANS PASSIVE TRANS)
        IMMOVABLEPARTICLES  ((IN BLOW~IN))
        N  -S
        V  ((PNCODE X3SG)
            (TNS PRESENT))]

[BLEW   V  (BLOW (PNCODE ANY)
                 (TNS PAST))]

[BLOWN  V  (BLOW (PASTPART))]
```

Nouns

```
[MAN    FEATURES (PASSIVE TRANS)
        N  IRR
        V  S-*ED]

[MEN    N  (MAN (NUMBER PL))]
```

Adjectives

```
[GOOD   FEATURES (FORTOCOMP SUBJCOMP SUBJHOLD THATCOMP)
        ADJ  IRR
        CASEPREPS   ((FOR) (WITH) (AT))
        N  -S]

[BETTER ADJ  (GOOD (COMPARATIVE))]

[BEST   FEATURES  (PASSIVE TRANS)
        ADJ  (GOOD (SUPERLATIVE))
        V  S-ED]
```

PRO

```
[I      FEATURES  (NOPOSTMODIFIERS)
        PRO  (I (SUBJ)
                (NUMBER SG))
        SEMANTICS  (PERSON *)]

[ME     FEATURES  (NOPOSTMODIFIERS)
        COMMENTS
           Maine abbreviation -donb, 5/30/86
        SUBSTITUTE ((MAINE))
        PRO  (I (OBJ)
                (NUMBER SG))

[YOU    PRO  (YOU (SUBJ)
                  (OBJ)
                  (NUMBER SG/PL))]
```

POSSPRO

```
[MY     POSSPRO  (I (POSS))]
```

Features describe the syntactic configurations in which a given lexical item can appear. For example, the feature **INDOBJFOR** indicates that a verb may be followed by an indirect object introduced by *for*. *Properties* are like features in that they restrict the distribution of particular lexical items; however, unlike features, they can take values. For example the property **UNIT**, which appears on *measure* nouns, such as *foot*, can have a value that indicates the dimension being measured, such as **LENGTH**:

```
[FOOT   FEATURES  (TRANS)
        N  IRR
        UNIT  (FOOT (UNIT LENGTH))
        V  S-ED]

[HOUR   N  -S
        UNIT  (HOUR (UNIT TIME))]
```

Some dictionary entries are associated with items that may appear as separate words in input but which should be treated as single units by the parser, such as *how long*. Such items are entered in the dictionary as single entries with the character ~ joining the parts. For example, *HOW~LONG* is the dictionary representation for *how long*. *Complex* verbs which consist of a verb and a particle are represented in the same fashion. For example, *SET~OFF* is the dictionary representation of *set off*.

Compounds

```
[NEW    ADJ  ER-EST
        COMPOUNDS
            (NEW JERSEY) => NEW~JERSEY
            (NEW YORK) => NEW~YORK]

[NEW~YORK    NPR  *]
```

```
[NEW~JERSEY   NPR   *]
```

Multiples

Verb+Particle

```
[BLOW~UP   FEATURES   (INTRANS PASSIVE TRANS)
           N   -S
           V   NOINFLECTIONS]
```

A dictionary editing tool is available to help the dictionary developer define new words and edit existing ones. The developer creates the dictionary definition of a word by selecting menu choices and typing in information when prompted. Another entry point to the dictionary interface is provided when the user attempts to use a word in a query that is not in the dictionary. The system asks the user if he wishes to define the word and if he selects yes, the dictionary interaction begins.

The dictionary tool is a window based tool which consists of a large interaction window and two menus containing word commands and dictionary commands. The word commands are **Add Word, Print Word, Edit Word, Delete Word**. These are discussed below. The dictionary commands are **Save Dictionary, Make Pretty Dictionary**. The dictionary commands are discussed in Section 5.6.

5.2 Add Word

When the dictionary developer selects the **Add Word** command, he is prompted for the name of the word he wishes to define. He is then presented with a menu containing the following options:

BREAK will place you in a LISP break. You should normally never use this option.

HELP will print the choices and also some information explaining them.

DESCRIBE will prompt you for the name of a part of speech and will print out a brief description of that part of speech.

NONE means that none of the current **CATEGORIES** apply. (There is a category called *SPECIAL* that should be considered instead of **NONE**, if you want to make that distinction.)

NEW-POS allows you to create a new part of speech not already on the **CATEGORIES** list. You would not normally use this option.

LIKE-OTHER-WORD
allows you to enter a second word. The word being defined receives a copy of the definition of that word. This is useful even if the definitions aren't exactly identical, since it might be easier to edit the copy than insert the complete definition from scratch.

<part of speech> The other possibilities are the parts of speech stored in the variable **CATEGORIES**.

For some parts of speech (e.g., *SPECIAL, CONJ, NPR*) only the category must be specified. For others (e.g., *N, V, ADJ*) a specific sequence of questions prompts for further information about the category. These options are described below.

Though the parser uses many categories, many of them are "closed classes" which you will not need to add to. The following categories are the only ones you are likely to actually input here: **ADJ, ADV, N(oun), NPR, V**.

Depending on the category which the user selects, the system presents the user with various prompts and menus to specify the characteristics of that word.

5.2.1 Nouns

Nouns are all uncapitalized (i.e. common) words which name a person, place, or thing.

If the user selects to define a NOUN, the system displays a menu asking how that noun is made plural. The choices are:

S for nouns like *word, dog, house, mountain*

ES for nouns like *box, tax, church*

IRR for nouns that form their plurals in an irregular ways, such as *ox/oxen, child/children*

BOTH FORMS THE SAME

PLURAL ALREADY

NONE

HELP This menu choice provides the user with information on the possible choices.

After the user has made his selection, another menu appears, asking whether the word has another plural form. If it does, the user is asked to select it from a menu.

The user is then asked whether the noun is non-countable. The HELP choice on this menu will provide the user with information on the meaning of this question. Non-countable nouns are concrete nouns that normally do not appear with a number (i.e., nouns that are not *count* nouns) and which can appear in singular form without a determiner. *Water* is an example of a non-countable noun. This feature is also used with nouns that are not concrete, such as *anger* and *freedom*.

The user is then given an opportunity to enter a comment into the new dictionary entry. The system will automatically create a comment that includes the date and the user name. Any comment the user enters will be added to this.

The user is then asked whether there are any substitution phrases involving the new word. The menu choices are Yes, No, Help and Quit. An example of a substitution phrase is the substitution of *Jan* for *January*.

5.2.2 Adjectives

Adjectives are words that modify nouns, e.g. *good, red, anxious*. Words that are past or present participles of verbs and might be used to modify nouns (e.g. *fallen, singing*) should *not* be classified as adjectives in the dictionary because the grammar will take care of them.

When a word is classified as an adjective, you must specify how it is inflected to get the comparative and superlative forms. Some *ADJ*s don't inflect at all (e.g. *only*).

Adjectives may take features. Those given only with examples below are described more fully in the section on verbs. The features **PREDADJ**, **SUBJHOLD**, **PREDETERMINER** and **POSTNOMINAL** are unique to adjectives.

PREDADJ The adjective can appear only in predicate-adjective position, not in prenominal position. For example: *the boy is afraid, *the afraid boy*.

PREDETERMINER
The adjective can occur before part of the determiner structure (numbers are considered part of the determiner structure), as in *the previous (last, biggest, next, most recent* four...).

POSTNOMINAL The adjective can occur after a noun, as in *the president elect*. Some of the words in this class occur only after a noun, others can also occur in the usual prenominal position.

CASEPREPS The adjective can take a noun phrase complement in the pattern <adj><p><np>. The **CASEPREPS** property allows the preposition to be specified for an individual adjective. e.g *she is afraid of him* (**of**), *John is faithful to Mary* (**to**), *they are too reliant on force* (**on**), etc.

THATCOMP as in *he is afraid that he will fail*

INTRANSTOCOMP
as in *she is careful to appear calm*

FORTOCOMP as in *it is difficult for him to hear*

SUBJLOW as in *he is eager to please* and *he is unlikely to come*

SUBJHOLD Adjectives which are marked **INTRANSTOCOMP** or **FORTOCOMP** may also have this feature, which means that the subject is held as the object of the complement rather than lowered into the subject position of the complement. (This construction is often called *tough movement* by generative linguists.) For example: *John is easy to please, Jim is difficult for Mary to please*

SUBJCOMP as in *that he will come is unlikely*

5.2.3 Adverbs

Adverbs are words that modify verbs, adjectives, or other adverbs. In addition to this part of speech, an adverb may have one or more of the following features: **PREDETADV**, **NEGADV**, **TRANSADV** and **ADJMOD**.

PREDETADV The adverb can occur before a noun phrase. (For example, *quite a woman, only a boy*.)

NEGADV The adverb cannot occur with a negated verb. (For example, *I ate the cake, I barely ate the cake, I didn't eat the cake, *I didn't barely eat the cake.*) In addition, adverbs with the feature **NEGADV** can occur in the sentence-initial position with subject-verb inversion, as in *Barely had I eaten...*

TRANSADV The adverb, while still modifying a verb, can have an object which is an NP, as in *especially the fat men*.

ADJMOD The adverb can modify adjectives, as in *almost red*, and *always delightful*.

5.2.4 Verbs

For verbs, you will be asked if the word you provided is the root form:

Is it the root form?

You will then be asked for the type of verb. The verb types are the following:

S-D if the word is a root inflected like *gauge*.

S-ED if the word is a root inflected like *learn*.

ES-ED if the word is a root inflected like *kiss*.

IRR if the word is an irregular verb like *go*, *be* or *see*; you will be prompted for the root of the verb and its conjugation

INFLECTED if the word is already inflected (like *gives* or *had*); you will be prompted for the other parts

When this information has been collected about a verb, information about the features of the verb will be gathered. Every verb must be transitive or intransitive (or both); transitive verbs may or may not take a direct object and may or may not be made passive. Explicit questions are asked for each of these features. If the verb takes particles, they may be either regular particles or immovable particles. Finally the system will ask **Does it take any complements?** If the answer is y, you will be allowed to choose one or more features from a list. (You can see the list by typing a ?.) In the future, more structure will be provided here, as some of the features aren't really complement features and some are dependent on others, but for now you can choose any subset. When you have finished entering features, type **END** to the ">" prompt.

Verbs are the most complex words to define because they may take so many features.

ASPECTVERB This is used for verbs that take infinitive or *-ing* complements (most take both, some take only one) and that have an *aspectual* interpretation. For example, *begin to read*, *begin reading*, *have to leave*, *stop trying*. Verbs marked with this property do not appear as verbs in the parse structure. Instead, they appear as features of the Auxiliary.

CASEPREPS The **CASEPREPS** property is used with adjectives and verbs that take (either optionally or obligatorily) a particular preposition, rather than prepositional phrases in general. The value is a list of pairs, the first element being the preposition, the second being a *semantic case*. An example of a possible value is **((TO RECIPIENT) (FROM DONOR))**.

ADJCOMP Verbs with this feature can take adjectives as complements. For example, *stand tall*.

TRANSADJCOMP
Similar to **ADJCOMP**, but the pattern is <subject><v><np><adj>. For example, *he drives me crazy*, *we find him guilty*.

IMMOVABLEPARTICLES
These are verb-particle pairs in which either there is no noun phrase object or the particle does not move after the object (as in *he washed up the dishes*, *he washed up*, **he washed the dishes up*). The pattern is <subj><v><particle><np> or <subj><v><particle>.

INTRANS Intransitive verbs can appear without an object noun phrase, e.g. *eat* and *die*.

INTRANSTOCOMP

These verbs (and adjectives) can only take a **to** complement without a preceding NP, as in *agree to buy, forget to call, threaten to do it.*

A verb can have only one of the features **TOCOMP**, **INTRANSTOCOMP**, and **NPTOCOMP**.

TRANS Transitive verbs can appear with an object noun phrase, e.g. *eat, give,* and *assist.* Verbs which are transitive may also have one or more of the following features:

PASSIVE The verb can appear with its deep structure object before the verb and its deep structure subject in a **by** phrase, as in *the rat was eaten (by the cat).* Many transitive verbs can be passivized but some cannot (e.g. *resemble, cost, weigh*).

INDOBJ The verb can take an indirect object, which appears with **to** or **for**. (The indirect object may or may not be required.) Usually there is a form without the preposition in which the indirect object comes before the direct object. For example: *John gave the book to him* and *John gave him the book.* There are a few verbs that appear only in the dative form (with *to* or *for*), e.g. *donate, explain.*

INDOBJ&THATCOMP

The verb can take a **that** complement together with an indirect object. For example: *I told her that he lied. Show him that it is raining.* The pattern is: <subj><v><NP>that<S> (In some cases the **that** is optional; in others it is required. See the discussion of the feature **THATREQUIRED** later in this section.)

INDOBJFOR The verb can take an indirect object which allows dative movement with the word **for**. (Again, the indirect object may or may not be required.) For example: *allow, find, make.* Note: verbs that have the feature **INDOBJFOR** cannot take **INDOBJ&THATCOMP**, too.

BITRANSITIVE The verb can take two noun phrases (one of which would seem to be an indirect object): the pattern is <subj><V><NP1><NP2> but not <subj><V><NP2>{to,for}<NP1>. For example, *they showed him a good time, *they showed a good time to him, he forgave us our sins,* and *we envy you your wealth.*

PARTICLES This is an indication that the verb to which this property is attached may take one or more particles. Each verb-particle pair is stored as a separate entry in the dictionary, since such a pair usually has a meaning quite different from the verb alone. Particles appear either before or after the direct object (as in *he cleaned up the mess, he cleaned the mess up*). A verb may take several particles, as in *give up, give in, give out.*

ITSUBJ-THATCOMP

This feature indicates a raising-to-subject verb, which either takes the word *it* in subject position together with a following **that** complement, or takes an infinitive (*to* + verb) whose subject appears as the subject of the original verb. *It seems that John is here. John seems to be here.*

IDQOBJECT The verb can take an indirect question word in the object position without making the whole sentence a question. For example: *I know who did it, They pondered over whether to do it* and *I asked the man how to get to Harvard Square.*

OBLIGATORYADV

The verb require an adverb, e.g. *he put it there, they meant well.*

The three features **PERCEIVECOMP**, **PARTICIPLECOMP**, and **BARE-INF-COMP** are mutually exclusive. In addition, a verb may have only one of the features **MAKECOMP**, **PARTICIPLECOMP**, and **PERCEIVECOMP**.

PERCEIVECOMP

Verbs with this feature can appear in the active form <V><NP><-ing or infinitive complement> and in the passive form <be><V>to<infinitive complement>. For example: *I heard John read(ing* the story) and *John was heard to complain.* Only a few verbs of perception, such as *hear, notice, see,* and *feel* have this feature.

PARTICIPLECOMP

Verbs with this feature must be transitive and may take a *-ing* or *-ed* complement. If no NP occurs after the verb, the complement must be *-ing*. For example: *I deny sending it, he saw her reading it, we had her treated at the hospital.*

BARE-INF-COMP
The only verbs with this feature are *have*, *let*, and *make*. The complement must appear with a preceding NP and a bare infinitive in the active form, and with *to* and the infinitive in passive form. For example: *Make him eat*, *He was made to eat*.

MAKECOMP Verbs with this feature act rather like transitive copulas. There are only a few in English, e.g. *declare*, *make*, *consider*. Some examples are: *She made him a good husband*, *I consider him a fool*.

NPTOCOMP The verb requires an object NP (which appears before the to complement in active sentences). For example, *I believe him to have eaten it*, *He order them to stop*, *John was believed to have died*.

SUBJLOW Verbs which are marked **NPTOCOMP** or **TOCOMP** may also have this feature, which means that the subject of the matrix sentence is always lowered into the complement to become the subject there. For example: *Mary promised John to leave early*.

THATCOMP The verb takes a **that** complement, such as *say (that) it is true*, *note that he failed*. Such verbs are usually transitive, but not always (e.g. *agree*).

THATREQUIRED
Some thatcomp verbs are also marked with this feature, which means that the **that** cannot be deleted from the complement (e.g. **he commanded they stop*). Some verbs with this feature will be borderline (*agree*); perhaps the grammar should be willing to relax this constraint. If this feature is absent but **THATCOMP** is present, assume that the **that** is optional.

SUBJUNCTIVEREQUIRED
Some **THATCOMP** verbs are also marked with this feature, which means that the verb in the embedded S is subjunctive. For example, *I demand that he be punished*. **I demand that he is punished*. ?*I demand my lawyer be present*. Verbs of this type seem to prefer their complements to contain a **that**.

COPULA The verb acts like the copula *be* in predicate adjective constructions, like *he {feels, appears, becomes, seems} sad*.

FORTOCOMP These verbs (and adjectives) allow **for to** complements in which the for and its NP are always optional. The paradigm is <V>(for NP)to<V>. For example: *I wish (for John) to die*, *I prefer (for him) to go.*.

Compare this feature with **INTRANSTOCOMP** and **TOCOMP**. Some verbs with the feature **TOCOMP** can look like **FORTOCOMP** verbs when they contain a long adverb in the middle (*I asked very loudly for him to leave*), but they should be marked only **TOCOMP**.

A verb may not be marked both **FORTOCOMP** and **INTRANSTOCOMP**; words which appear to be both should be marked **FORTOCOMP**.

NOLOW Verbs which are **FORTOCOMP** take this feature if the subject is *not* to be lowered to become the subject of the lower verb. For example, *He said to go*.]

SUBJCOMP This feature may appear on **INTRANS** verbs and some *ADJ*s which allow a complement in the subject position. Often this subject is replaced by *it* and the complement appears later. For example: *for him to pass the test is unlikely*, *it is unlikely that he passed the test*, *that he will pass it is unlikely*, *to pass the test is hard*, *it is hard to pass the test*.

TOCOMP This feature applies to verbs which optionally allow an NP before a **to** complement. For example: *Ask (them) to leave*, *I'd like (him) to sit down*, *He chose (her) to go*. (If an NP is required, the verb should be labeled **NPTOCOMP**; if it is never allowed, it should be labeled **INTRANSTOCOMP**.) Note that in some cases, a long adverb before the first NP results in what looks like a **FORTOCOMP**: *I asked very loudly for him to leave*; **NPTOCOMP**s never do this. Mark these verbs as **TOCOMP**s, not **FORTOCOMP**s. Verbs with **TOCOMP** may also be marked **SUBJLOW**.

It is very important to distinguish all forms of **to** complements from the construction which substitutes *to* for *in order to*. Most sentences allow an **in order to** construction an adjunct (modifier), as in *he ran to catch the train*,

and the grammar handles these. A verb should be assigned a **to** complement feature only if the examples do not allow **the in order to** substitution.

5.2.5 Other Categories

BINDER These are subordinating conjunctions, such as *although*, *if*, *while* and *because*.

COMP These are words used in comparative constructions, such as *at least* (*AT~LEAST*), *less*, *nearly*.

CONJ These words are conjunctions for simple constituents such as NP's. For example: *and*, *as well as* (*AS~WELL~AS*), *but*, *unless*, *or*, *yet*.

DET Determiners are words that can occur before adjectives in an NP. Words that other systems might classify as quantifiers or articles are *DET*s in the IRUS-II system (e.g. *a*, *all*, *both*, *no*, *that*, *the*, *these*, *what*, *which*).

INTEGER These are number words, such as *seven*, *eighteen*, *forty*. (NB: the class of integers is not closed, but the class of words used to express the integers is, at least for our purposes.)

INTENSIFIERADV
means that the word can modify an adverb, as in *most quickly*, *very quickly*, *too quickly* and *really quickly*.

MONTH These are the months, *January* through *December*. (Abbreviations such as *Jan* and *Feb* are handled with a **SUBSTITUTE** property.)

UNIT These words are nouns which are also units of measurement, such as *foot*, *year*, *degree*, *gallon*, *tablespoon* and *meter*. The value of this property gives the root form of the unit together with the feature **UNIT** and an optional value indicating the dimension of that unit. Thus the **UNIT** property of the word **FEET** is **(FOOT (UNIT LENGTH))** while that of **DOLLAR** is **(DOLLAR (UNIT))**. The dimension is needed if the unit participates in constructions such as *ten feet long*, *a meter wide* and *three minutes long*. Currently recognized values for this property are: **LENGTH**, **TIME**, **VOLUME**, **TEMPERATURE**.

Words like *long*, *wide*, *full*, and *farenheit* have the features **TIME**, **LENGTH**, **VOLUME**, and **TEMPERATURE** respectively to indicate that they can be used in phrases such as *3 feet long*, *2 hours long*, *2 teaspoons full*, *9 feet deep*, *17 degrees Farenheit*.

NEG The only entries in this class are *not* and *n't*.

NPR Proper nouns are nouns such as *Apollo*, *China*, *Washington*. Proper nouns are (usually) not modified by adjectives and (usually) do not take determiners.

DETREQUIRED This is for proper nouns that *do* take a definite determiner, such as *the Hague*, *the White House*, *the Netherlands*. The grammar will allow proper nouns with this feature to be used without a determiner if they function as modifiers, for example, *White House spokespeople agreed ...* (There are a few *NPR*s such as *Earth* that can appear with or without a determiner, but the case without can probably be treated as a plain noun rather than as an *NPR*.)

ORD These are the ordinal numbers, e.g. *first*, *second*, ...

POSS The only words in this category are ' and 'S.

POSSPRO These are the possessive pronouns *my*, *your*, *his*, *her*, *our*, *their*, *its*, etc.

PREP Prepositions introduce prepositional phrases which can modify verbs, nouns, or adjectives. (Some *PREP*s are: *about*, *ahead of* (*AHEAD~OF*), *up*, *for*). Prepositions can take three features:

BAREPREP This feature means that the preposition can occur alone without an object NP, for example, *below*, *down*, *here*, *inside*, *there* and *up*. (Some dictionaries consider these words to be adverbs, but we disagree.) Some bareprep can optionally take a noun phrase as their object, for example, *walk out*, *walk out the door*.]

NONPOBJECT This features indicates items that are treated as *PREP*s by the parser which cannot take an NP object. Such items include *here*, *there*, etc.

PPOBJECT These prepositions can take whole prepositional phrases as their objects, as in *here from the city*, *across from John's house*, and *near to the lake*.

PRO Pronouns are all the non-possessive and non-question pronouns, such as *anyone*, *he*, *her*, *I*, *me*, *myself*, *that* and *it*. Pronouns may have the feature **NOPOSTMODIFIERS**.

NOPOSTMODIFIERS The pronoun cannot participate in constructions such as *she who I heard from* and *he who laughs last*.

PUNCT These are punctuation marks, such as '.', ',', ';', '?' and others.

QDET These words are question determiners: *how many* (*HOW~MANY*), *how much* (*HOW~MUCH*), *what*, *which* and *whose*.

QPRO These are the question pronouns *what*, *who*, *whom*.

QUANTADJ These are the quantifier adjectives *how many* (*HOW~MANY*), *many*, and *several*.

QWORD These are the question words *how*, *what time* (*WHAT~TIME*), *when*, *where*, and *why*.

SENTCONJ These conjunctions can conjoin entire sentences, such as *and*, *but*, *nor*, *or*, *though*, *until*, and *yet*. A word may be both a **CONJ** and a **SENTCONJ**.

SHORTANSWER These words can be used alone as answers to questions or as introductory words to a sentence. For example: *never mind* (*NEVER~MIND*), *no*, *yes*, *oh* and *ok*. For our purposes, this is closed class.

SPECIAL These are a few words that simply don't belong in any of the other categories, such as *than* and *etc*.

TITLE For example: *Mr*, *Ms*, *Dr*, *Chief Justice* (*CHIEF~JUSTICE*), *sir*.

5.3 Edit Word

The user may choose to edit an existing word. This may either be a word that was added during the current session, or a word that is already in the dictionary. When the user chooses Edit Word from the dictionary menu, he is prompted for the name of the word he wishes to edit. The words added during the current session are checked first. The files that were opened during system initialization are then tried one at a time until the word is found. Thus, a more recently opened file takes precedence over a previously opened file. The purpose of this mechanism is to allow patch files that override an existing file. If the word does not have a dictionary entry, the system displays a message to that effect. If there is a dictionary entry for the word, the current definition of the word is displayed on the screen. The user is then presented with a number of choices that can be selected from a menu.

Want to Check, Edit, Replace, Delete, Add, Help, File or Ok?

The options here are:

- **Check**: checks the consistency of the current syntactic features of *word*. If they are consistent, the message

  word **is alright.**

  will be displayed. If there is some inconsistency, the following message will be printed out:

Problems with *word*:

followed by a specification of the inconsistency. The editor checks for the following forms of inconsistency:

Presence of a feature that is only associated with a particular category:
for example, only an *ADJ* can have the feature **PREDADJ**. If a word that is not an *ADJ* has this feature, the following type of message would be printed out:

((*word* **PREDADJ requires ADJ**))

Presence of a feature that requires another feature:
for example, only *V*s that have the feature **TRANS** can have the feature **PASSIVE**. If a word has the feature **PASSIVE** without the feature **TRANS**, the following type of message would be printed out:

((*word* **PASSIVE requires TRANS**))

Occasionally, a feature is contingent on one of several features. For example, **THATREQUIRED** can appear on a word that has one of the features **THATCOMP** or **INDOBJ&THATCOMP**. If a word that does not have one of these features has the feature **THATREQUIRED**, the following type of message would be printed out:

((*word* **THATREQUIRED requires-one-of (THATCOMP INDOBJ&THATCOMP)**))

- **Edit**: Use the **Edit** command only if you are sure you understand the internal representation of the kind of property you are changing. Use **Delete** and **Add** otherwise.
- **Replace**: prompts for the replacement value for the specified property.
- **Delete**: is used to remove one of the existing properties of *word*. Removes the SPECIFIED property from the dictionary entry.
- **Add**: is used to add a property. You will be prompted for the value of the property just as if you were defining that part of *word* for the first time. Adds the specified property to the dictionary entry.
- **Help**: prints a help message for the specifed property.
- **File**: exit and save the dictionary entry in dictionary. **File** and **Ok** both indicate that you are satisfied with the definition of the word and wish to exit from the word editor, but File indicates that you want *word* to be filed immediately.
- **Ok**: exit and leave the dictionary entry. Indicates that you are satisifed with the definition of *word*, but that you do not want to file it now.

Each time you make a modification to *word*, the definition will be redisplayed along with the question, **Want to Check, Edit, Replace, Delete, Add, Help, File or Ok?**

5.4 The Compounds, Multiples, and Substitute Properties

The following properties are not restricted to appearing on particular parts of speech and can appear on any dictionary entry.

5.4.1 Compounds

The COMPOUNDS property is attached to words which begin compound phrases. The compound phrase should be an entry in the dictionary.

```
(UNIT
  COMPOUNDS ((REPORT (UNIT~REPORT))
             (IDENTIFICATION NIL (CODE (UNIT~IDENTIFICATION~CODE))))
  N  -S
  SEMANTICS  (UNIT *))))
```

5.4.2 Multiples

The MULTIPLES property is attached to words whose meaning can be expressed with an alternate phrase

```
(PACIFIC MULTIPLES ((FLEET (PACIFIC~FLEET))
                    (OCEAN (PACIFIC~OCEAN))))
```

5.4.3 Substitute

The SUBSTITUTE property indicates that the word in question should be replaced by the given substitute word. This is used for abbreviations (e.g. **Jan** for **January**) and contractions (e.g. **can't**).

```
(JAN      SUBSTITUTE ((JANUARY)))
```

5.5 Print Word

The selection of **Print Word** on the dictionary menu enables the user to display the definition of a word on the screen. The user is prompted for the word to be displayed. If the word is undefined, the system displays a message to that effect.

5.6 Dictionary Commands

The two dictionary commands on the dictionary menu are **Save Dictionary** and **Make Pretty Dictionary**.

5.6.1 Save Dictionary

The **Save Dictionary** command saves the words into dictionary files.

5.6.2 Make Pretty Dictionary

The **Make Pretty Dictionary** command enables the user to get a version of the file that people can read. Keep in mind that since each word has to be read from the dictionary file and written to the pretty file, making a pretty file can be time consuming.

5.7 Semantics

Chapter 4 referred to the fact that dictionary entries link words to the corresponding domain model concepts. In some of the examples in the current chapter, **SEMANTICS** has appeared as one of the features of the example words. In this section we briefly discuss the process of assigning SEMANTICS to dictionary entries. The motivation for assigning specific SEMANTICS features is to be found in the domain model and the IRULES developed for the domain.

To add SEMANTICS to a word, the user selects **SEMANTICS** from the menu listing categories and features. This menu is displayed during the process of adding or editing a dictionary entry. The user then selects **ADD** from the next menu. The user is prompted for the SEMANTICS to be added. The SEMANTICS property is a list of pairs associating semantic categories (concepts or roles in the domain model, or names of irules) and restrictions on those categories. First the category is entered; then the restriction. Usually there is no restriction, so a value of * is entered. The following examples illustrate unrestricted semantic properties of some words.

```
(SUBMARINE   N -S
             SEMANTICS (SUBMARINE *)

(CARRIER     N -S
             SEMANTICS  (CARRIER.VESSEL *)

(DEPLOY      FEATURES  (BITRANSITIVE INTRANS PASSIVE TRANS)
             SEMANTICS  (DEPLOYMENT *)
             V  S-ED)
```

A review of these examples indicates that there are few surprises in the assignment of SEMANTICS to a word. The word *SUBMARINE* is represented in the IRULEs as the domain model concept **SUBMARINE**. The word *carrier* has the SEMANTICS **(CARRIER.VESSEL *)** associated with it. The semantic interpretation of *carrier* is thus dependent on the treatment of **CARRIER.VESSEL** in the domain model and the IRULEs. Likewise, the verb *deploy* is semantically characterized as **(DEPLOYMENT *)**.

The following are examples of words whose SEMANTICS properties have restrictions on the semantic categories:

```
(FREDERICK  FEATURES  (DETREQUIRED)
            NPR  *
            SEMANTICS  (CONVENTIONAL.SURFACE.SHIP (NAMEOF "Frederick")))

(HONOLULU   NPR  *
            SEMANTICS  (MULTIPLE-SEMANTICS
                        ((PORT (NAMEOF "Honolulu")
                          NUCLEAR.SUB (NAMEOF "Honolulu")))))
```

Frederick happens to be the name of a ship. The SEMANTICS property specifies this with the expression **(CONVENTIONAL.SURFACE.SHIP (NAMEOF "Frederick"))**. *Honolulu* has two semantic categories associated with it, because it is both the name of a port and the name of a nuclear submarine. Each semantic category has a **(NAMEOF "Honolulu")** restriction.

6. World Model Language Representations of Queries

6.1 Intensional Logic

One of the difficulties in developing a system of logical representation for natural language is that traditional logic is much more limited in its expressive range than natural language. Intensional logic represents one direction in broadening the expressive range of logic so that it covers a broader scope of natural language.

Intensional logic, as implemented at BBN, extends the capabilities of extensional logics by allowing explicit reference to times and possible worlds. For example, it can represent propositions whose truth values vary depending on time (a characteristic typical of real-world applications); in addition, it provides for propositions which are either true in some possible world, or necessarily true in all possible worlds.

The INTENSION operator makes this possible. The INTENSION of an expression represents a function taking two arguments: a time specification, and a world specification. Given these specifications (which can be thought of as contextual indices), the function returns the denotation of the expression at the time and world specified. So, for example, the intensional expression INTENSION(alive(shakespeare)) represents a function that takes a time argument and a world argument, and returns a truth value. With arguments *July 1, 1988* and *current-world*, the value returned is FALSE; with arguments *July 1, 1600* and *current-world*, the value returned is TRUE. Terms, as well as propositions, may have intensions: for example, INTENSION("the person reading this sentence") represents a function which, given the current instant and the current world, denotes you, the reader.

In many cases, querying an underlying system can be thought of as extensionalizing a description given by the user. Part of the WML representation of "Which carriers are C1" represents the intension of the set of C1 carriers, and it is the system's task to extensionalize this, returning the set denoted by the expression, in this case in the current time and world.

In general, intensional logic's specification of time and world permits the representation of complex natural language utterances. This new logic is powerful enough to express generics, collective quantification, modalities, tense, propositional attitudes, mass terms, and discourse context.

In practice, of course, many queries do not require the use of explicit contextual indices. In such cases, the time and world arguments default to *TIME* and *WORLD*, which represent the current time (time of the utterance) and current world.

6.2 Domain Model Constants and Predicates in WML

In addition to branch-categories such as SETOF and INTENSION, and logical constants such as TRUE and NULL-SET, WML comprises elements from the domain model. The roles in the domain model correspond to binary predicates in WML; for example, the domain-model role *VESSEL.NAME* corresponds directly to the binary WML predicate *vessel.name(x, y)*, which is a relation taking two arguments of the same types as the domain and range of *VESSEL.NAME* (namely, vessels and strings). The concepts in the domain model correspond to unary WML predicates, or, equivalently, to the positive extensions of those predicates. For example, the domain-model concept *VESSEL* corresponds to the WML predicate *vessel(x)*, or to the corresponding set *{x/vessel(x)}*.

When one expands the domain model, therefore, one is also extending the World Model Language, by adding new unary and binary predicates. Section 4.2 discusses how IRules are used to compose WML.

6.3 Example

Here we present an example of a query's representation in WML, pointing out its relationship to the domain model, and explaining some aspects of its construction.

Query: "Which ships are in the Indian Ocean?"

WML expression generated by IRUS-II:

```
(QUERY
  ((INTENSION
    (PRESENT
      (INTENSION
        (IOTA ?JX9 (POWER VESSEL)
          (IN.PLACE ?JX9 INDIAN.OCEAN)))))
   TIME WORLD))
```

VESSEL and *INDIAN.OCEAN* correspond to concepts in the domain model; *INDIAN.OCEAN* is an individual concept, and for this reason is treated as a constant in the WML expression. The binary predicate *IN.PLACE* corresponds to a role in the domain model -- a role whose domain includes vessels, and whose range includes the Indian Ocean (were this not the case, semantic constraints stated in the Irules would have been violated, and this WML expression never produced).

The central subexpression in this query is

(IOTA ?JX9 (POWER VESSEL) (IN.PLACE ?JX9 INDIAN.OCEAN)).

IOTA introduces a scoping expression, and as such introduces a *sorted* variable, ?JX9 in this case, which is

constrained to denote an element of the set of all possible sets of VESSELS, indicated by (POWER VESSEL) in the *sort field* of the expression. The *body* of the IOTA expression further constrains the possible values of ?JX9, to require that the IN.PLACE relation holds between it (?JX9) and INDIAN.OCEAN. An expression with branch category IOTA denotes an individual, or set of individuals, and indicates that only element of the set that its variable ranges over is salient, i.e., in this example, only one maximal SET of VESSELS is expected to be a possible instantiation for ?JX9. (Note that this is not Russel's *iota*, which is denotationless if more than one instantiation is possible: in WML, if a IOTA variable ranges over a power set, as in this example, the IOTA denotes the maximal set that the variable can range over (the maximal set of VESSELS in the INDIAN OCEAN). If the variable ranges over a simple set, only one possible instantiation is expected for the variable, and more than one (or none) indicates a presupposition violation.)

Wrapping the INTENSION operator around this subexpression, we have an expression that represents a function from times and worlds to sets of vessels; the PRESENT operator partly constrains the function, requiring that the time-index argument be within "the present" (however that is defined) in order to have a value returned. Given that the TIME and WORLD arguments immediately follow, what we really have is

(QUERY <*extension*>),

where <*extension*> denotes the set of ships in the Indian Ocean in the current time and world. The speech-act (QUERY) indicates that the extension's value(s) should be returned to the user.

6.3.1 WML Types

The formal syntax and semantics of WML are in [11]. In this and the following section, we present an overview of the type system and the definition of meaningful WML expressions.

Each valid WML expression has a branch category, usually the CAR of the expression. For example, the branch category of

```
(EXISTS ?JX1 UNIT (UNITS.OVERALL.READINESS ?JX1 C1))
```

is EXISTS. Branch categories are the "glue" that make complex WML expressions out of simple expressions. Each WML expression has a corresponding TYPE. The extension of a WML expression must be a member of the WML's type. For example, the type of the expression with branch category EXISTS above is TV (for truth value), and it must denote either TRUE or FALSE. The type of

```
(IOTA ?JX2 VESSEL (NAMEOF ?JX2 "Frederick"))
```

is VESSELS, and it denotes a member of the set of VESSELs.

A WML expression is meaningful if it has a TYPE that is *not* NULL-SET. (NULL-SET a special type; the type of meaningless WML expressions.) Determining the type of an expression is a recursive process, it is a function of the types of its parts. Each branch category has a corresponding *type constructor* that determines how the type of an expression with that branch category is obtained from the types of its parts. By providing a principled

way to obtain the type of complex WML expressions, we are given a mechanism to test the validity of an expression and the range of its possible denotations.

The set of possible types TYPE is defined (recursively) as:

- TV, INT, T, and W (for truth value, integer, time, and world, respectively), are in TYPE.
- Whenever a domain model concept C denotes a set of individuals, C^T is in TYPE.
- Whenever ta, tb are in TYPE, then (ta tb) is in TYPE.
  (ta tb) is the type of a function whose domain is ta and range is tb, i.e., one can construct arbitrarily complex functions.
- Whenever ta1,ta2,...,tan are in TYPE, then (TUPLE-TYPE ta1,ta2,....tan) is in TYPE.
- Whenever ta1,ta2,...,tan are in TYPE, then (UNION-TYPE ta1,ta2,....tan) is in TYPE.
- Whenever ta is in TYPE, then (BAG-TYPE ta), (SET-TYPE ta), and (ORDERED-BAG-TYPE ta) are in TYPE.
- NULL-SET is in TYPE, it is the type of denotationless expressions.

6.3.2 Meaningful WML Expressions

This section formally defines the set of meaningful WML expressions, by defining all valid WML branch categories, and giving type constructors for them. Any expression not satisfying the following definition is denotationless (has type NULL-SET).

Note:

- We use the convention that parentheses and atoms in CAPITAL LETTERS should be taken verbatim, as quoted expressions, but atoms in small letters denote meta-variables which stand for arbitrary expressions.
- TYPE-OF is a lisp function which, when given an input x, "returns" the type of x if x is well-formed, and NULL-SET otherwise. We use the notation := for "returns". (TYPE-OF x) := tx means, "when (APPLY #'TYPE-OF x) is executed, it returns tx".
- Speech acts are in our language, and are used in the interpretation of a sentence. They are special branch categories. The WML corresponding to an utterance will always have the form (speech-act ((INTENSION x) time world). The set of speech acts includes QUERY, ASSERT, and BRING-ABOUT.

1. (TYPE-OF variable) := tv, where tv is the sort in the expression that defines the scoping environment for the variable.
2. (TYPE-OF *T*) := T, (TYPE-OF *W*) = W. *T* and *W* are special variables in WML.
3. (TYPE-OF domain-model-role-constant) := ((TUPLE-TYPE td tr) TV) where td and tr are the domain and range types of the role in the domain model.
4. (TYPE-OF (LAMBDA (u) p b)) := (tu tb), if u is a variable of type tu, (TYPE-OF p) := (tu TV), and (TYPE-OF b) := tb.
5. (TYPE-OF (f a)) := tr, if (TYPE-OF f) := (td tr) and (TYPE-OF a) := ta, and the intersection of td and ta is non-empty.
6. (TYPE-OF (EQUAL a b)) := TV, if (TYPE-OF a) := ta, (TYPE-OF b) := tb, and the intersection of ta and tb is non-empty

7. (TYPE-OF (PRED-TO-SET p)) := (SET-TYPE ta), if (TYPE-OF p) := (ta TV).

8. (TYPE-OF (CARD s)) := INT, if (TYPE-OF s) := (SET-TYPE ta).

9. (TYPE-OF (SET-OF a1 a2 ... an)) := (SET-TYPE (UNION-TYPE ta1 ta2 ... tan)), if (TYPE-OF a1) := ta1, (TYPE-OF a2) := ta2,..., and (TYPE-OF an) := tan.

10. (TYPE-OF (FORALL u p b)) := TV, and (TYPE-OF (EXISTS u p b)) := TV, if u is a variable of type tu, (TYPE-OF p) := (tu TV), and (TYPE-OF b) := TV.

11. (TYPE-OF (IOTA u p b)) := tu, if u is a variable of type tu, (TYPE-OF p) := (tu TV), and (TYPE-OF b) := TV.

12. (TYPE-OF (TUPLE-OF n a1 a2 ... an)) := (TUPLE n ta1 ta2 ... tan), if (TYPE-OF a1) := ta1, (TYPE-OF a2) := ta2, ..., (TYPE-OF an) := tan.

13. (TYPE-OF (UNION p)) := (ta TV), if (TYPE-OF p) := ((ta TV) TV).

14. (TYPE-OF (POWER p)) := ((ta TV) TV), if (TYPE-OF p) := (ta TV).

15. (TYPE-OF (APPLY-COLLECT f s)) := (BAG-TYPE tr), if (TYPE-OF s) := (SET-TYPE ta) or (TYPE-OF s) := (BAG-TYPE ta), (TYPE-OF f) := (td tr), and the intersection of td and ta is non-empty.

16. (TYPE-OF (MEMBER a s)) := TV, if (TYPE-OF s) := (SET-TYPE ta) and (TYPE-OF a) := ta,.

17. (TYPE-OF (BAG-TO-SET b)) := (SET-TYPE tb), if (TYPE-OF b) := (BAG-TYPE tb).

18. (TYPE-OF (FIRST n o-bag)) := (ORDERED-BAG-TYPE ta) if (TYPE-OF n) := INT and (TYPE-OF o-bag) := (ORDERED-BAG-TYPE ta)

19. (TYPE-OF (SORT set fun)) := (ORDERED-BAG ta) if (TYPE-OF set) := (SET-TYPE ta), (TYPE-OF fun) := (td tr), and the intersection of td and ta is non-empty.

20. (TYPE-OF (ORDERED-BAG-TO-SET o-bag)) := (SET-TYPE ta) if (TYPE-OF o-bag) := (ORDERED-BAG-TYPE ta).

21. (TYPE-OF (SKOLEM n p)) := ta if (TYPE-OF n) := INT, and (TYPE-OF p) := (ta TV)

22. (TYPE-OF (AND f1 f2)), (TYPE-OF (OR f1 f2)), (TYPE-OF (IMPLIES f1 f2)), (TYPE-OF (IFF f1 f20)), (TYPE-OF (NOT f1)), (TYPE-OF (NECESSARILY f1)), and (TYPE-OF (POSSIBLY f1)) := TV, if (TYPE-OF f1) := TV and (TYPE-OF f2) := TV.

23. (TYPE-OF (INTENSION a)) := (W (T ta)), if (TYPE-OF a) := ta.

24. (TYPE-OF (EXTENSION i)) := ta, if (TYPE-OF i) := (W (T ta)).

25. (TYPE-OF (<tense-operator> i))) := ta, where <tense-operator> is one of PRESENT, PAST, FUTURE, PRESPERF, and PASTPERF, if (TYPE-OF i) := (W (T ta)).

26. (TYPE-OF (G i)) := (ta TV), if (TYPE-OF i) := (W (T (ta TV))), and ta is a type of individuals (a domain concept type).

27. (TYPE-OF (SAMPLE i)) := (ta TV), if (TYPE-OF i) := (W (T (ta TV))), and ta is a type of individuals.

28. (TYPE-OF (KIND-OF i)) := ta, if (TYPE-OF i) := (W (T (ta TV))), and ta is a type of individuals.

29. Nothing else is a WML expression.

7. Users' Guide to KNACQ in IRUS-II

7.1 Purpose of KNACQ

This chapter describes the component KNACQ (KNowledge ACQuisition) [33] of BBN's IRUS-II natural language system. This component greatly eases the task of defining the lexical semantics for words in a new domain. KNACQ assumes a predefined domain model that describes in the NIKL Knowledge Representation language the logical concepts (unary predicates) and roles (binary predicates) that are relevant to the domain. KNACQ annotates that domain model with information about how those concepts and roles can be referred to in natural language. It acquires that information for each concept and role from a system developer.

KNACQ records various kinds of data about the natural language expressions related to each concept in the domain model. The simplest kind of knowledge acquired by KNACQ is the NL terms that can be used to refer directly to instances of the concept, for example, that instances of the vessel class "CG" can be referred to as "cruisers".

Much of KNACQ's power comes from a second class of data that captures and exploits the common elements found in the many different ways in which the relational nouns that describe attributes of classes can be used in database contexts. For example, KNACQ acquires from the developer the information that the "speed-of" role connecting "vessels" to "speed-measures" can be referred to by the relational noun "speed". Using that data, the names for the concepts, and the fact that "speed-measures" are defined as "one-dimensional-scalars" in the domain model, KNACQ then makes it possible for IRUS-II to correctly interpret a great variety of expressions referring to vessel speeds, including:

- the speed of the cruisers
- the vessel's speed
- the Vinson has a speed of 20 knots
- vessels with a speed of less than 20 knots
- the fastest vessel

KNACQ thus makes available a great deal of linguistic power based solely on the names given to concepts and roles.

Other portions of KNACQ cover phenoma that do not fit the attribute pattern. For example, a more general scheme called *caseframe* rules handles lexicalizations of roles that use different prepositions or reference patterns than apply to attributes in general. For example, suppose that a "casualty report" can be said to be "from" a "military unit", where the prepositional object specifies the filler of the "unit-of" role for the "casualty report" concept. Note that this example inverts the usual pattern from prepositional phrases embodying relational nouns: "the unit of report-36", like "the speed of Frederick", is the normal relational case, where the object of the

prepositional phrase is the primary concept, the domain of the role in question, but in "the report from unit-12", the main noun gives the primary concept, while the object of the preposition specifies the relationship by specifying its filler. Caseframe rules provide for such more general lexicalization patterns for roles.

Special KNACQ code also handles those gradable adjectives whose meaning is derived from the value of a single role, like "fast", in the sense of having a greater speed than another vessel or than a given cutoff value.

The KNACQ data acquired from the system developer is stored as annotations attached to the NIKL domain model. When that model is in turn loaded into IRUS-II, those annotations are converted into semantic fields in the definitions of the given words that allow IRUS-II to parse sentences containing them. The KNACQ acquisition process is highly interactive, and is implemented as an extension to the KREME knowledge base editing system, which allows the developer to explore the NIKL network graphically and edit it incrementally. Note, however, that the connection between KNACQ and IRUS-II is currently implemented by having KNACQ write out a complete version of the domain model, including the KNACQ annotations, and then having IRUS-II read and process that file. Thus, domain model changes or KNACQ additions will not take effect in IRUS-II until that file writing/reading is done. There are plans to make this connection more fully incremental in the future.

While the various facilities of KNACQ go a long way toward speeding up the acquisition of lexical semantics for new domains, they do not do the whole job. KNACQ is attribute treatment handles that portion of noun phrase semantics that depend on highly-predictable patterns. Though caseframe rules allow for coverage of simple non-attribute NPs, the semantic entries for more complex noun phrase elements and for verbs still need to be entered directly, by creating IRULES for each sense and attaching those IRULES to the words involved. The IRACQ component of IRUS-II provides an example-based scheme for acquiring verb semantics. Note that KNACQ's initial coverage of noun phrase semantics contributes in turn to using IRACQ for verb phrase semantics, since it provides a wide range of noun phrases for use in the example sentences that IRACQ uses. Together, KNACQ and IRACQ allow rapid bootstrapping acquisition of the semantics for new domains.

7.2 Adding KNACQ Data to the Domain Model

The KNACQ interface for adding or changing the data associated with a given concept or role is implemented as supplementary functionality added to the KREME knowledge representation editing system. (The files implementing these additions are termed the KUI system, for KNACQ User Interface.) To use KNACQ, use Select-K to access KREME, mouse the "parameters" button in the "main commands" menu and select KNACQ as the "language mode". (KNACQ mode in KREME is a submode of NIKL mode; if it does not seem to be listed as an option, make sure that the NIKL-COMPATABILITY and KUI systems are loaded.) Then choose the main command "load network", entering the names of the file or files containing the NIKL domain model. (To enter more that one file name, type a single comma by itself after each file name except the last.) After the network has been loaded and classified, the KREME command "edit concept", given a concept name, will bring up three

windows: a graph, showing that concept with its ancestors and descendents, a window describing that concept's definition, and a window listing the roles for which that concept is either the domain or a subconcept of the domain. What you will actually see in response to "edit concept" is a small box, indicating the system's request for a location for the new collection of three windows. Moving the mouse will move the box around; moving the mouse while the shift key is held down will move just the lower right corner, allowing resizing of the space for the three new windows. These windows can also be resized and moved after being displayed through menus accessible by mousing right on their top label lines.

Following the usual KREME conventions, mousing right on either the icon or name for a concept or role pops up a menu of operations on that object; one of those options in turn requests a menu of KNACQ operations. Concepts can be grabbed in this way from the graph or the description windows; roles are most easily accessed in the list of slots window, but note that the mouse must be on the name of the role itself, and not on the whole row in the table.

While in KNACQ mode, the KREME display is modified to indicate the presence of KNACQ data attached to roles and concepts, to make it easier to see when the domain model has been fully covered by KNACQ data. For concepts, the necessary KNACQ data is limited to natural language terms that can be used to refer to instances of that concept; an elliptical icon is used for concepts that have such KNACQ vocabulary defined, while the normal KREME rectangular icon is used for concepts with no vocabulary. Roles may carry not only vocabulary information referring to their attributive use but also the more general caserule specifications for other kinds of natural language reference to the given role and specifications for gradeable adjectives based on the role. Information on the KNACQ data attached to roles is included in the list of roles (called "slots" in KREME parlance). Columns there show the number of vocabulary items defined as attribute names for the role, the number of adjectives, and whether or not caserule specifications are defined. While the list of slots window shows all slots, both those locally defined at this concept and those inherited from a superconcept, the columns showing what KNACQ data has been defined only apply to slots that are defined locally. To see that display for inherited slots, move up instead to editing the concept at which they are defined.

A user wishing to enter KNACQ data for a new domain model may choose to do so in any order. One reasonable approach is to move from concept to concept through the network, adding both concept vocabulary for each concept and role information for each of the roles attached to it, working in this way through all the concepts. Concepts not yet handled are obvious, since they display as rectangular-shaped icons. Mousing-right on a role in the list of slots window brings up a menu of commands to add to or modify its KNACQ data. As an alternative, the command "KNACQ process concept" in the KNACQ menu for concepts will go through the steps for acquiring first vocabulary for the concept and then attribute, caserule, and adjective information for all the roles attached at that concept.

Because the slots display window shows counts of the KNACQ data, its contents are normally redisplayed each time the user adds to or changes the KNACQ data for a slot. The same holds for the window where the definition of the current concept is displayed, since it displays the now changed internal form of the KNACQ data as user attached data. If this redisplaying proves annoyingly slow, setting the flag **kui::*refresh-p*** to **nil** will cause the slot's window's contents to be refreshed only when the entire window is redisplayed.

The following sections describe in more detail the different classes of data KNACQ requires.

7.2.1 Terms for Concepts

For each concept, KNACQ can record a set of terms (usually common nouns) that can be used to refer to entities belonging to that class. For example, the concept "destroyer-class-vessel" might carry KNACQ data noting that any instance of that class can be referred to as a "destroyer" and perhaps also as a "tin can" (Navy slang). Because NIKL inheritance also specifies that "destroyer-class-vessel" is a subclass of "vessel", KNACQ realizes that terms like "ship" and "boat" that are attached to the concept "vessel" can also apply to "destroyers".

For each such term, KNACQ also records inflectional information showing how it can be made plural. KNACQ will accept a set of vocabulary items, terminated by an empty carriage return, and it then queries for the inflection of each item. Setting the flag **kui::*plural-query-p*** to **nil** will suppress the user queries about inflection, leaving KNACQ to depend on its own heuristics for guessing the correct plural. KNACQ allows the use of multiple word entries, like "tin can" for destroyer, although the facility is not fully general, since it assumes that inflection affects only the last word in the sequence.

7.2.2 Terms for Attribute Roles

The primary information stored by KNACQ about roles is the vocabulary used to describe their use as attributes. For example, the "speed-of" role connecting "vessels" to "speeds" can be referred to by the attribute terms "speed" and "velocity".

Note that KNACQ information can be attached to slots either (as is typically done) at their defining concepts or (when desired) at a concept that merely inherits the slot. The latter is appropriate when an NL term can be used to describe a slot only in relation to a particular class of entities. For example, while all "physical-objects" might be defined to have a "width" slot, the term "beam" is used to describe that slot only when speaking of ships; thus "beam" would be attached as a KNACQ term for the "width" slot at the concept "vessel". Because there is this option of attaching KNACQ information to slots at concepts that inherit rather than define them, KNACQ operations on slots first ask for a concept name to specify the concept with respect to which this term is to be used. The default answer for that question is always the defining concept, which is usually also the appropriate answer.

With the related concept established, the user can then enter attribute terms for the slot. Inflectional information is collected for these attribute terms, in the same way as for concept vocabulary.

KNACQ also makes it possible to record families of attribute names, using "attribute modifiers", for cases where adjectives modifying the name cause it to refer to a different role. For example, "speed" by itself might act as a name for the role "vessel-speed-of" while the modified names "maximum speed" and "design speed" act as names for the separate roles "vessel-maximum-speed-of" and "vessel-design-speed-of". KNACQ allows such

modified roles to be entered, along with the roles to which they actually refer. (Note, however, that the modifying words like "maximum" or "design" need to be themselves defined syntactically as words for this facility to work. KNACQ was not designed to do this automatically because it was not clear that their syntactic type could be safely guessed in all cases. Also note that such modifiers are stored at the role named by the main attribute term, "vessel-speed-of" in the example, and not at the roles that the modified attributes end up referring to.)

7.2.3 Caseframe Patterns for Roles

While the attribute style covers a large percentage of the ways in which role data can be referred to in NL, a more general facility in KNACQ known as "caseframe rules" allows the system developer to select from a menu exactly the appropriate set of prepositional phrases or nominal compounds by which the given role can be referenced. One example requiring the more general caseframe approach was given above, where "the report from unit-12" did not fit the pattern of "the report's unit". In that example, the role as defined in the domain model is in the opposite direction from the way it is normally referred to. Thus, we are presuming here that the domain model includes a role from casualty reports to the ship that filed them. Attribute names could be used with that role to handle references like "the ship of report-112". However, in examples like "the casualty report of the Fox" the attribute rule approach would take the vessel as the main noun, though the role here is attached to the report. Thus caseframe rules, which can apply in either direction, must be used in order to connect phrases like that with the underlying role from reports to ships.

Adding a caseframe rule is one of the options on the menu attached to roles. Mousing that option brings up a substantial menu of possible forms of reference to the given role. For example, if the role were the one connecting ships and speeds, the menu would include the noun phrases "the ship speed" and "the speed ship" along with many possible prepositional phrases, including "the ship among the speed" and "the speed among the ship", "the ship upon the speed" and "the speed upon the ship", and the like. Most of these possible expressions will not actually be ways of referring to that role in English, but some will, and the user should select the ones that both sound right and could reasonably be interpreted as referring to the given role. For prepositional phrases, the KNACQ menu also makes a distinction between those that can only occur as part of the main NP and those that can also occur as a predicate. As an example of the former type, the phrase "the commander of Frederick" works, while "the commander is of Frederick" does not; as an example of the latter, "the Admiral aboard the ship" and "the Admiral is aboard the ship" are both OK.

7.2.4 Gradeable Adjective Terms

Certain domain model roles like the speed of a vessel refer to scalar functions where the values of that role for different instances of the concept can be compared like grades on a scale. KNACQ allows the developer to enter adjectives that can refer to such roles, like the adjective "fast" for speed. For each gradeable adjective, KNACQ asks for the root form, the inflectional information ("fast, faster, fastest"), and whether "X is faster than Y" means that X's value for this role is greater than or less than Y's.

Such gradable adjectives can also be used in an absolute sense, as in the phrase "a fast ship". KNACQ implements a straightforward scheme whereby the developer can enter a single threshold value for each adjective stating the limit above or below which instances of that concept or subconcepts can be said to satisfy such absolute uses of the given adjective. The user may enter more than one sense of "fast", with different concepts as their domain to indicate, for example, that "a fast battleship" is faster than "a fast landing craft".

7.3 Making KNACQ Data Available in IRUS-II

KNACQ stores the information it learns about NL ways of referring to particular concepts and roles by attaching data to the :DATA slot of the concepts and roles. When the user then saves the network, the KNACQ data is saved along with the normal NIKL definitions of the concepts and roles. (If the network is reloaded into KREME/KNACQ in a later session, this attached data is used to reestablish the state that existed when the network was last saved.) When this annotated form of the NIKL network is loaded into IRUS-II, this attached data is loaded along with it. The function (process-domain-model-knacq-data) then uses that attached data to derive IRUS-II syntactic and sematic entries for words in the IRUS-II dictionary.

This process of transferring data acquired by KNACQ to IRUS-II currently works only for the entire domain model at once. Further changes made in KNACQ will not become effective in IRUS-II until the next time the entire network is saved out from KNACQ and reloaded into IRUS-II.

8. Access to Multiple Underlying Systems in Janus

8.1 Introduction

As computer systems become more complex, there is more opportunity for combining the strengths of more than one system in order to perform a task. For example, one might imagine combining several resources: a database for storing relational information with an applications program to perform calculations based on that information, an expert system to perform inferences, and a display system to present data in a useful way. In such an environment a "seamless" natural language interface can become a very effective tool, allowing the user to retrieve and manipulate information without needing to pay attention to the details of any particular resource.

The back-end of such an interface, however, is necessarily more complex: not only must it be able to translate the user's request into executable code, but it must also be capable of organizing the various resources at its disposal, choosing which combination of resources to use, and supervising the transfer of data among them. We call this the *multiple underlying systems* (MUS) problem. This chapter describes the critical information needed for the implemented *MUS component* that is part of the back end of the Janus natural language interface.[1]

8.2 The Type System

The syntax of WML is "modelled after languages of the typed lambda calculus" ([10], p. 27). The import of this is perhaps most concisely expressed in [29]:

> Each type expression [or *type*] is associated with a set of entities or structures which is termed its *domain*. Every expression of the language... can be mapped to a type expression, whose domain serves to delimit the range of values the expression can take on.

A complete description of the type system associated with WML is beyond the scope of this document, but in this section we attempt to convey a sense of it by means of examples.

The set of types has two major subsets -- those that are independent of the domain, and those that are specific to the domain. The former set includes the types TV (truth-value), INTEGERS, STRINGS, REALS, TIMES, and WORLDS; the latter consists of types constructed from concepts and roles in the domain-model. For example, if *VESSEL* and *LOCATION* are domain-model concepts, and *SHIP-LOCATION* and *COMBAT-READY* are domain roles, then some related types might include:

[1] This chapter focuses on the information that must be provided in applying Janus to a new application and is an abbreviated version of [29] which provides more detailed documentation of the MUS component.

Type	Denotation
VESSEL	all vessels
(S VESSEL)	all *sets* of vessels
(TUPLE VESSEL LOCATION)	all ordered pairs of vessel, location
(S (TUPLE VESSEL LOCATION))	all sets of such ordered pairs
(FUNC-TYPE VESSEL LOCATION)	functions from vessels to locations
(FUNC-TYPE VESSEL TV)	unary predicates on vessels

Here we present some types together with examples of logical subexpressions having those types (that is, *(TYPEOF expression) = type).*

1.
```
type VESSEL
             Vincennes
             (IOTA ?JX1 VESSEL (COMBAT-READY ?JX1))
```

2.
```
type (S VESSEL)
             (SETOF Vincennes Kennedy)
             (SET ?JX2 VESSEL (COMBAT-READY ?JX2))
             (IOTA ?JX1 (POWER VESSEL) (COMBAT-READY ?JX1))
```

```
type (FUNC-TYPE VESSEL LOCATION)
             SHIP-LOCATION
             (LAMBDA (?JX3) VESSEL (VESSEL-LOCATION ?JX3))
```

```
type (FUNC-TYPE VESSEL TV)
             COMBAT-READY
             (LAMBDA (?JX4) VESSEL
               (AND (EQUAL (VESSEL-LOCATION ?JX4) "HAWAII")
                    (COMBAT-READY ?JX4)))
```

8.3 Normalizing WML Expressions

WML input expressions are simplified and normalized before they are further processed by the multiple underlying systems (MUS) component. This simplification process has five stages:

1. the extensionalization of intensional subexpressions (for further information, see [20]),
2. the translation of the entire expression into a modified disjunctive normal form,
3. the elimination of unnecessary equivalences (for further information, see [20]),
4. the application of underlying-system-independent rewrites, and
5. the use of printfunctions for improving responses.

These stages occur sequentially, and -- as the system is currently implemented -- must be done in the order presented here. Stages 2, 4, and 5 are discussed further below.

8.3.1 Disjunctive Normal Form

The second stage of normalization is the translation of the extensionalized WML expression into a somewhat simplified logical expression in a modified disjunctive normal form (DNF).

The expression is translated into a disjunctive normal form for two main reasons. We normalize the expression (reducing the number of embedded subexpressions, for example) in order to simplify the process of matching various pieces of it to underlying system capabilities. We choose to use a disjunctive normal form because:

- In the simplest case, an expression in disjunctive normal form is simply a conjunction of clauses, a particularly easy logical form to cope with.
- Even when there are disjuncts, each can be individually handled as a conjunction of clauses, and the results then combined together via union, and
- Bringing disjunctions to the top level allows patterns to match in many cases where it would otherwise not be possible. For example, given the (non-normalized) pattern

```
(AND   (OR   (IN.CLASS ?JX1 SUBMARINE)
             (IN.CLASS ?JX1 AIRCRAFT))
       (LENGTH ?JX1 ?JX2))
```

a service seeking to match the pattern

```
(AND (IN.CLASS <x> SUBMARINE)
     (LENGTH <x> <y>))
```

could not match. The DNF, on the other hand,

```
(OR   (AND (IN.CLASS ?JX1 SUBMARINE)
           (LENGTH ?JX1 ?JX2))
      (AND (IN.CLASS ?JX1 AIRCRAFT)
           (LENGTH ?JX1 ?JX2)))
```

allows the match to take place, by keeping the relevant information together. In a disjunctive normal form, each disjunct effectively carries all the information necessary for a distinct subquery.

A standard disjunctive normal form is a disjunction of conjunctions of predicates or negated predicates; no variables in such an expression are explicitly quantified, and all are assumed to be implicitly universally quantified. Existentially quantified variables have been replaced by skolem terms denoting some individual instantiation of the variable.

The modified disjunctive normal form differs from a standard DNF in several respects:

- There is a *response clause* for every query; that is, an additional predicate whose arguments are the variables for which we want returned values. In a query requesting a set of objects (e.g. "Which ships are in the Indian Ocean?") the argument in the response clause will be the variable denoting the set in question. The same is true for queries requesting individuals (e.g. "the ship whose speed is 30 knots"); the resulting logical form will seek *all possible* individuals that meet the same description.

  In a yes/no or existential query, the response clause will contain all variables in the query, since any instantiation of all the query variables means an affirmative answer; the inability to find any such instantiation means an answer in the negative. In such cases, a particular variation on the response predicate is used: rather than using the special predicate *RESPONSE*, we use the special predicate *VALUE-EXISTS-RESPONSE* instead; this preserves the information that the query's intent is to find out

if values exist, rather than to have them returned.[2]

- All functional terms must appear as predicates: if P is a binary predicate and Q is a unary function, then *P(x, Q(y))* must appear as *P(x, z) and Q'(y, z)*, where Q'(y, z) is true iff Q(y)=z. For example, the clause

```
(GREATER-THAN (SPEED-OF ?JX1) 30)
```

will appear as

```
([AND] (SPEED-OF' ?JX1 ?JX2)
       (GREATER-THAN ?JX2 30))
```

- Similarly, provision is made for complex terms like database aggregates; for example, cardinality, average, and sum. Such complex terms may only appear as the first argument to a special predicate called *IS-TERM*; the second argument is always a variable that represents the term. For example, if the logical expression asks for the cardinality of the ships in the Indian Ocean, we would use the following clause:

```
(IS-TERM #S(CONTEXT
             :OPERATOR CARDINALITY
             :OPERATOR-VAR ?JX2
             :CLASS-EXP
               ((IN.CLASS ?JX2 SHIP)
                (SHIP-LOCATION ?JX2 "INDIAN OCEAN")))
           ?JX1)
(RESPONSE ?JX1)
```

- There is no implicit assumption of universal quantification for unquantified variables -- expressions in the modified DNF may contain universal quantification.

- Existential quantifiers are removed, not by replacing existentially quantified variables with skolem terms, but simply by removing the explicit existential quantification. The resulting unquantified variables, along with all other unquantified variables in the form, are considered to be *query quantified*.

  The term *query quantified* refers to variables for which we would like to get all possible instantiations. Such variables are neither existentially quantified (since we're interested in all instantiations) nor universally quantified (since universal quantification has no notion of returning values); this kind of quantification is more like that of the variables in a PROLOG expression.

  Notice that, because universal quantifications are not removed, there is no need to skolemize existentially quantified variables appearing within the scope of universal quantifiers.

8.3.2 System-independent Rewrites

At this stage of the normalization process, the system permits the application of obligatory rewrite rules. These rules must be independent of the underlying systems: both pattern and result must consist of domain-model information, and they may not contain any references to structures or data in the underlying system(s).

Rewrite patterns may seek to match both simple clauses (i.e., those that are not contexts), and context-clauses. Similarly, results may be either contexts or simple clauses. For example, the following rewrite might be used if it was known that the number of subordinates of a manager corresponded to the number of employees in a manager's department:

[2] A cooperative system may still return the value -1 if they exist; for example, "Are any ships C1" might lead to the response *Yes, the C1 ship are*

```
(define-simple-rewrite[3]
  :pattern ((in.class x manager)
            (:context
             :operator CARDINALITY
             :class-exp ((in.class y person)
                         (subordinate-to y x))))
  :result  ((in.class x manager)
            (department-of x z)
            (employee-count z y)))
```

By using this rewrite rule, we transform a query in which one actually counts elements in a set (via the cardinality term) into one in which a single table lookup is used instead.

8.3.3 Printfunctions

Often the logical content of a query does not reflect its desired interpretation. For example, a query as simple as "List the cruisers," if interpreted literally, produces a listing of the database's *internal representation* for each cruiser. In the Navy's IDB domain, this representation is a number called an IUID -- a number that is almost certain to be completely useless to the user as a means of ship identification. What one would *really* like is for the system to be smart enough to interpret the question as "List the *names* of the cruisers." *Printfunctions* provide just that functionality.

The printfunction machinery is quite simple. With certain classes of objects (e.g., the domain-model concept *VESSEL*) one associates a specification for how members of that class should be presented to the user, called a *printfunctions list*. Each element of the printfunctions list (each *printfunction*) is either (1) the name of a domain-model role, or (2) the special symbol :IDENTITY. As a postprocessing step of the normalization, the variables on the response list are examined, and a *new* response list created.

Printfunctions are inherited -- in the examples in the following section, responses involving ship classes like cruisers and aircraft-carriers are always expressed as responses involving the names of the ships because the class *VESSEL* (the top-level class for ships) has the printfunction list *(NAMEOF)* associated with it.

[3] In simple rewrites, x, y, z, and w are always variables.

8.4 Servers and Services

In an environment with multiple underlying systems, one must have a uniform way to describe the capabilities of each underlying system. We adopt terminology similar to that of [12] and [17].

A *server* is a functional module typically corresponding to an underlying system or a major part of an underlying system. In the application of the MUS system being described here, there are two servers -- one named :ERL, which supports access to a relational database, and one called :LISP, which supports calls to arbitrary LISP functions. Each server has associated with it:

1. A number of *services*: objects describing a particular piece of functionality provided by a server. Specifying a service in MUS provides the mapping from fragments of logical form to fragments of underlying system code
2. An *execution planner*: a function that takes a piece of the *solution* to a query and builds from it a *partial execution plan*
3. An *executor*: a function that takes a partial execution plan together with input data, executes the plan, and produces output data (see section 8.5).

A service is an object consisting of the following components:

- **Name**: a symbol used to uniquely identify the service
- **Owner**: name of the server to which this service belongs.
- **Cost**: a scalar value indicating the cost of this service; if unspecified, unit cost (1) is assumed.
- **Inputs**: a list of pattern variables, each of which has associated with it a name, a type, and a constraint. The type indicates the extent to which the input is optional: a type of *:GEN* indicates that input to this variable is optional, since this service can generate values for the variable; a type of *:TEST* indicates that input must be provided for the variable, since this service is only capable of applying some test to the input values; a type of *:TEST-ALL* indicates not only that input to this variable is obligatory, but that by the time the data for this variable reaches this service it must be filtered as completely as possible -- this is often the type for inputs to services that do response presentation, for example.

  The constraint associated with the variable is used for pattern-matching. The possible constraints include:

  1. *(symbol+)*: a list of symbols. Items matching this variable must be EQ to a symbol on the list.
  2. *(string+)*: a list of strings. Items matching this variable must be STRING= to a string on the list.
  3. *type*: a simple type. An item will match this variable if the type (i.e., type-system type -- see section 8.2) of the item is a subtype of *type*. (The type of a Janus variable *v* is *type* if the clause *(IN-CLASS v type)* appears in the (normalized) query.) This constraint does not pay attention to whether or not a type denotes a set -- if *type* is *(S SHIP)* (a set of ships), an item with type *SHIP* will match, and vice-versa.
  4. *function*: a function than takes one argument. An item *item* will match this variable if *(funcall function item)* returns a non-NIL value.
  5. *(SUBTYPE-OF type)*: a subtype specification. Items matching this variable must themselves *be types in the type system*; furthermore, they must be subtypes of *type*. For example, a variable with constraint *(SUBTYPE-OF SHIP)* would match *CRUISER* (since *CRUISER* is a subtype of

SHIP). Notice how this differs from (3), above: there the constraint is that *(SUBTYPE (***TYPEOF** *item) type)* must be true, whereas here the constraint is that *(SUBTYPE item type)* must be true.

6. *NIL, ANYTYPE, T*: these will match anything.

- **Outputs**: a list of pattern variables, identifying the outputs of the service. Outputs need not have been inputs, nor must inputs to the service also be outputs.
- **Pattern**: a pattern specification which will match some piece of the logical form. The pattern specification must be a list, each element of which is the pattern for either a simple clause or for a context-clause. Within patterns, one can not have a variable predicate; however, the arguments to predicates *must* be pattern variables (see *inputs*, above, for a description of how to constrain what these variables may match).

  A pattern specification for a context-clause (*context-spec*) takes one of two forms.

```
(:context :operator operator
          [:free-vars (var+)]
          [:operator-var var]
          [:stat-var var]
          [:class-exp expression]
          [:constraints expression])
```

  For this form of pattern specification, the context-spec's operator must match the context's operator, and recursive calls to the matcher must return successfully for the :class-exp and :constraints.

```
(:context :operator operator
          [:covers-owner server-name])
```

  This second form of pattern specification allows one to say, "This service will match *any* context whose operator is *operator*, as long as there are solutions of the :class-exp subexpression and of the :constraints subexpression such that both solutions belong entirely to server *server-name*." For example, a context-spec for operator *CARDINALITY* specifying that it covers owner *:ERL* says, in effect, "This service can take the cardinality of any set, as long as that set can be obtained entirely by calls within the :ERL server." This is a useful method of providing general services that handle aggregate operations within a single server.

- **Method**: a code fragment or other information that the server will use in generating a partial execution plan from a solution that utilizes this service. What goes in the method slot depends entirely on the particular server to which the service belongs.

For fast access, services are indexed by the predicates in their pattern. That is, for every clause *(P x y)* in the pattern of some service S, there is a pointer from the symbol *P* to the service S. An exception to this is the *IN.CLASS* predicate: if a service's pattern includes *(IN.CLASS x C)*, the pointer will be from the symbol *C* rather than the symbol *IN.CLASS*; that is, the indexing proceeds as if the clause were *(C x)*.

As an example, consider the service-object corresponding to the :ERL server's ability to access a table associating ships with overall combat-readiness values:

```
NAME:        VESSEL-OVERALL-READINESS-OF859
OWNER:       :ERL
INPUTS:      (<x> <y>)
OUTPUTS:     (<x> <y>)
PATTERN:     ((VESSEL-OVERALL-READINESS-OF <x> <y>))
COST:        NIL[4]
METHOD:      (((VESSEL-OVERALL-READINESS-OF X Y))
               (BINDTOERL ((X IUID) (Y RDY)) IID.RDY))
```

The name and owner fields are straightforward: the service has a unique name and belongs to the server named :ERL (in a current implementation, :ERL is the server that can access the Navy's relational database). The pattern is also particularly simple, a single clause. Note that the variables printed as <x> and <y> are objects:

```
NAME:        X
TYPE:        :GEN
CONSTRAINT:  VESSEL

NAME:        Y
TYPE:        :GEN
CONSTRAINT:  READINESS-RATING
```

Because both are type :GEN, this service does not require input values for these variables. The pattern will match clauses only when the type of the first argument (matching <x>) is VESSEL, and the type of the second argument (matching <y>) is READINESS-RATING.

The method field for services belonging to the :ERL server contains two pieces: first, the pattern that was matched; second, a code-like fragment that relates variables to fields and specifies a table from which to draw those fields.[5]

The scheme for indexing services establishes a pointer from the symbol VESSEL-OVERALL-READINESS-OF to this service.

8.5 Execution

The execution phase takes an execution plan (i.e., a list of partial execution plans), and iterates through it sequentially:

```
For each partial execution plan p
  Combine the data from the streams in the wrapper of p
  Call the execution function for the owner of p
  Pass the output tuples (according to the dataflow links of p)
   into the wrapper objects of partial plans further on
Return the output provided by the last partial execution plan
```

[4]The cost field is unspecified, therefore this service is assumed to have unit cost.

[5]This is a simplification: the table specification may be a fragment of ERL code, complete with JOINs, SELECTs, etc.

The execution function (or *executor*) for a server is a function taking the following arguments:

1. A list of tuples representing input values,
2. A sequence of Janus variables identifying the tuple elements, and
3. Code produced by the execution planner.

The execution function should return two values:

1. A list of tuples representing output values, and
2. A sequence of Janus variables identifying the tuple-elements.

8.6 An Example Backend

Since relational databases are typical services, we describe one here briefly. The previous generation of IRUS made use of a set of rules to translate from MRL expressions to ERL (extended relational language) expressions [18]. Although IRUS-II makes use of WML rather than MRL, and although there is now an additional component (the multiple-systems component) between the logical form and the underlying system, the MRL-to-ERL translation rules (MRLRules) still serve much the same purpose as before: they relate domain-model concepts and roles to the relations in the underlying database. As such, they are used essentially without modification in IRUS-II.

They do, however, serve an additional purpose. The multiple-systems component makes use of these rules in order to generate its model of the capabilities of the underlying database. This process has been automated, and consists primarily of creating a "service" for each rule that describes its input and output characteristics.

MRLRules are the mapping from the domain model concepts and roles to Extended Relational Language. Each MRLRule provides the IRUS system with the information it needs to translate one domain model object into an SQL-based database access language.

The function that installs one of these rules is DEFMUSRule. It has two arguments, a form specifying the domain model object you are providing the translation for, and the translation.

```
(DEFMUSRule (<dm node name> &rest args)
  (BindToERL ((Arg1 att1 ...) (Arg2 ...) ...)
    <An ERL relation>))
```

The <dm node name> is a symbol that has a concept or role definition in the domain model. The number of arguments provided will effect the ways that this rule can be applied during translation. There are generally 2 arguments. If there is only one, the rule can only be used to translate a single argument predicate. If there are 2 or more arguments, the N-argument rule can be used to translate an N-ary predicate, an (N-1)-ary function with 1 result, or an (N-1)-ary predicate. Binary predicates and unary functions are the most common case.

For example,

```
(DEFMUSRule (FOO a b) (BindToERL ((a fooA) (b fooB1 fooB2)) FooTable))
```

defines the translation for the domain model object named FOO. It enables the system to translate FOO as a binary predicate as in (and (foo v1 v2) ...) or as a function of one argument as in (GT (foo v1) v2) or as a unary predicate as in (and (foo v1) ...). The first enforces the restriction that the relation FOO exists between variables v1 and v2. The second allows the system to generate more complicated predicates and expressions involving the FOO relation. The third is a class restriction. It ensures that the FOO relation exists between v1 and some other object. IRUS will probably only use one form of expression for each domain model object but the MRLRules allow translations of any of the three.

The example above says that when translating a predicate (foo v1 v2), v1 is represented in the database by the field fooA in the table FooTable, and v2 is represented in the database by the fields fooB1 and fooB2 in the table FooTable. To enumerate the domain of the FOO predicate, you would project FooTable over fooA. To enumerate the range of the FOO predicate, you would project FooTable over the set fooB1 and fooB2. This is all you need to tell the system. IRUS then has all the information it needs to use expressions involving FOO in a translation.

In a Navy domain, the underlying system that extracts data from the IDB (integrated database) is called :ERL. For "List the ships", the following WML expression is produced by the front-end.

```
(BRING-ABOUT
  ((INTENSION
     (EXISTS ?JX1 LIST
        (OBJECT.OF ?JX1 (IOTA ?JX2 (POWER VESSEL) T))))
   TIME WORLD))
```

The normalized expression produced by the back-end is:

```
(AND (IN.CLASS ?JX1 LIST)
     (IN.CLASS ?JX2 VESSEL)
     (OBJECT.OF ?JX1 ?JX2)
     (MEMBER ?JX1 ?JX3)
     (IN.CLASS ?JX3 (POWER EVENT))
     (RESPONSE ?JX3))
```

After rewrites and printfunctions have applied, the output of the MUS component is:

```
((RESPONSE ?JX70)
 (NAMEOF ?JX2 ?JX70)
 (IN.CLASS ?JX2 VESSEL))
```

To answer the question "List the ships", the NAMEOF and IN.CLASS relations are combined and the result is sent to the response generator. The translation rules used are:

```
(DEFMUSRule (NAMEOF x y)
  (BindToERL ((X IUID) (Y SHIPNAME)) IID.UCHAR))

(DEFMUSRule (Vessel X)
  (BindToERL ((X IUID)) IID.UCHAR))
```

Note that (IN.CLASS X Vessel) is equivalent to (Vessel X).

The two ERL fragments are joined and then optimized. The resulting SQL statement is "Select IUID, SHIPNAME From IID.UCHAR." The results are then made available to other systems by placing them in the wrapper for the partial execution plan.

Appendix A
Installation Instructions

The IRUS-II system consists of two modules, called the Front End and the Back End. The Front End takes an English sentence as input and returns a logical representation of that sentence in the World Model Language (WML). The Front End contains the grammar, lexicon, interpretation rules, and domain model.

The Back End takes a WML as input and transforms it into one or more commands to underlying systems, executes those commands, and combines the results from the Underlying Systems.

The various modules and systems that make up the Front and Back ends are changing as BBN's Natural Language technology develops. The following section details a snapshot of the current IRUS-II system. The names of these systems are subject to change. In order to install the FCCBMP version of IRUS-II, the procedure is to load a distribution tape (using the **Read Distribution** command processor command) and then to do Load System for each of the major systems included in the IRUS-II system, as outlined below.

A.1 Systems Included in IRUS-II

IRUS-II currently consists of the following major systems, each of which also may include component subsystems:

- The systems CLOS, BASIC-NEWKREME, CONDITIONS, NEWKREME, NEW-WINDOWS (which includes basic-grapher and clustered-windows), and WINDOW-EDIT together provide the KREME knowledge editing environment, on which the KNACQ acquisition system is based.
- The system IRUS-II (which includes isi-clisp, isi-nikl, clisp-records, and janus-front-end) represents the core of the natural language parser with support code.
- JANUS-KNACQ, NIKL-COMPATABILITY, and KUI adjust the parser to work with the KNACQ knowledge acquisition system, and load the domain model and dictionaries.
- The system EMBEDDED-GRAPHER provides for graphing parse trees.
- The system PARAPHRASER (which includes mumble86, text-planner-core, parrot-core, and parrot-navy-knacq) adds the generation capabilities.
- MUS-FOL-JANUS (which includes basic-back-end and mus-fol-basic), ERL-MUS, and ERL-INTERP make up the back-end components, which translate from the internal meaning representation to a modified first order logic (FOL) and direct the computation in the multiple underlying systems (MUS) to get the answer.
- The JANUSKA-I system (which includes irus-i) provides the frames and windows for this version of the IRUS-II interface.

A.2 Installation Instructions

BBN distributes software in two formats:

- IDS files on FEP tapes
- Binaries and/or sources on Distribution Tapes

This section describes how to load software from each of these distribution formats.

A.2.1 Loading IDS Files From FEP Tape(s)

A world made of IDS files consists of one or more "layers". Each IDS file is a "layer" of core image. In order to boot a world made from IDS files, you must first make sure all the required IDS files and the correct microcode are on your machine, and that you have a correct boot file.

BBN cannot distribute Symbolics software, including operating system distribution loads and microcode files. You must get them from Symbolics.

A.2.2 Do You Have The Correct IDS Files?

The FEP maintains a database of IDS files. The following commands operate on that database:

- **Clear IDS Files** clears the database,
- **Find IDS Files** searches the FEP for IDS files, adding them to the database
- **Show IDS Files** displays the database
- **Add IDS File** adds an IDS file to the database.

NOTE: These IDS commands allocate memory within the FEP as they run. This memory is never deallocated, so the FEP runs out of memory on occasion. When this happens, you have to reset the FEP.

To look at your IDS files, use the command processor command Show Fep Directory :type World. This will display all the IDS files on the FEP, indented as to generation. (See the Symbolics manual for interpretation of the display.) In order to boot a world containing an IDS file, you need all the ancestors of that IDS file.

If you are missing ancestors, either you didn't read all the files from the FEP tapes BBN sent you, or you have to get operating system or microcode files from Symbolics.

A.2.3 Resetting the FEP

When the FEP runs out of memory, use the command **Reset Fep**. This will spin down the disks and clear the FEP's memory. If the video image on your console loses sync (i.e. you can't read it and you get nothing but snow or moving bands of black), that means you have the wrong monitor type setting. There are two types of monitors, so try resetting the monitor type with the following commands:

- **Set M M**
- **Set M P**

One of these should work. After resetting the FEP and setting your monitor type, use the **Hello** command to initialize the FEP and spin up the disk.

A.2.4 Do You Have The Correct Microcode?

The correct microcode file for a band has the correct *version number* for the band you are booting, and the correct *file name* for the hardware configuration you are running on. To determine the correct *file name*, look at a boot file on your machine that boots successfully. When you boot a world, the FEP prints a message containing the loaded microcode version number and the version number expected by the band being booted. NOTE. It is possible to load the right microcode version number for the wrong hardware configuration. The world might successfully boot and later die a mysterious death. For further information on microcodes, see the Symbolics Manuals.

A.2.5 Reading FEP Tapes

The IRUS-II system requires a Symbolics machine with Genera 7.2 already installed and with sufficient blocks free in its FEP. The number of blocks required for each system is written on the tape containing that system. If your machine has no tape drive, you will have to read the tape on another machine that does have one and then transmit the bands to your machine. We will use the terms **destination machine** and **tape drive machine** to refer to these two machines.

The **FEP Tape** program is used to read FEP tapes. Use the command **Select Activity FEP-TAPE** to select the FEP Tape program.

Once the FEP-Tape screen appears, you are ready to read the tape. If the destination machine and the tape drive machine are the same (the machine you are currently using has a tape drive), then you just need to execute the following command:

```
Read Tape :Full Length Tapes Yes
```

If the destination machine and the tape drive machine are different, then execute the following command:

```
Read Tape :Full Length Tapes Yes :Host tape host machine
```

For each file on the tape, the **Read Tape** command prompts with the name of the file, allowing you to specify either where to put the file or to skip reading the file. Just use the default name that you are presented with.

For more information on the **Read Tape** command for the **FEP-Tape** facility, see the *Site Operations* (vol 0) manual of the Symbolics documentation set.

A.2.6 Editing The Boot Files

Now you must create a file irus-ii.boot. Boot files have machine specific information in them, including the hardware configuration (in the name of the microcode file) and the network address of the machine. Boot files copied from another machine must be edited before use! Whenever you do this, you'll at least have to change the network address, and perhaps the microcode file, if it's a different machine configuration.

A.2.7 Booting the BBN NL System

Type the following to the FEP:

```
Boot irus-ii.boot
```

Because the band is being booted at a site other than the site at which it was built, the machine will ask you if the site is still BBN. Answer **NO** and the machine will name itself **DIS-LOCAL-HOST**.

If the machine has identity problems (thinks it is still at BBN), the simplest way to deal with them is to unplug the ethernet before booting. See your local system wizard if you want a more elegant solution.

If you want your machine to regain its correct identity, you can use the **Set Site** command to tell it that it is at your site, instead of at BBN. You need to know the chaos address of your namespace server machine. See the Symbolics documentation about this command.

A.3 Loading Carry Tapes

In some cases, you may receive a tape with demonstration files or other individual files on it. These are usually sent on a tape format called a *Carry Tape*.

To load a carry-tape, use the command

```
(tape:carry-load)
```

You will have to choose a place on your machine to store any files loaded from a carry-tape.

A.4 Transferring FEP Files Onto Other Machines

If you want to have the IRUS-II software on more than one machine, you can use the **Copy World** command processor command to copy the FEP world files onto other machines. You can also use the **si:transmit-band** and **si:receive-band** functions.

For instance, if you want to transfer FEP file from the machine you are currently logged into to another machine, you can evaluate the following:

(si:transmit-band "fep0:>*file-name***.load"** *destination-machine*)

After you copy the FEP files, you need to make sure your destination machine has an irus-ii.boot file.

A.5 Loading a Distribution Tape

To load a distribution tape, use the **Read Distribution** command processor command. When loading distribution tapes, it is not necessary that the target machine have a tape drive. The load-distribution-tape command asks what machine the distribution tape is in, and reads the tape across the network.

Once the files have been loaded onto the destination machine, use the Load System command. The correct form of Load System will be listed on the tape label.

References

[1] Abrett, G., Burstein, M., Gunshenan, J., and Polanyi, L. *KREME: A User's Introduction.* Technical Report 6508, Bolt Beranek and Newman Inc., 1987.

[2] Abrett, G and Burstein, M. The KREME Knowledge Editing Environment. *Int. J. Man-Machine Studies* 27:103-126, 1987.

[3] Ayuso, D. Discourse Entities in Janus. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 243-250. 1989.

[4] Bates, Madeleine. Accessing a Database with a Transportable Natural Language Interface. In *The First Conference on Artificial Intelligence Applications*, pages 9-12. IEEE Computer Society, December, 1984.

[5] Bobrow, Robert J. *The RUS System.* BBN Report 3878, Bolt Beranek and Newman Inc., 1978.

[6] Bobrow, R.J. and Webber, B.L. Knowledge Representation for Syntactic/Semantic Processing. In *Proceedings of The First Annual National Conference on Artificial Intelligence*, pages 316-323. American Association for Artificial Intelligence, 1980.

[7] Brennan, Susan E., Friedman, Marilyn W., and Pollard, Carl J. A Centering Approach to Pronouns. In *Proceedings of the 25th Annual Meeting of the ACL*, pages 155-162. ACL, 1987.

[8] Grosz, B., Joshi, A.K., and Weinstein, S. Providing a Unified Account of Definite Noun Phrases in Discourse. In *Proceedings of the 21st Annual Meeting*, pages 44-50. Association for Computational Linguistics. Cambridge, MA, June, 1983.

[9] Grosz, B., and Sidner, C. Attention, Intentions, and the Structure of Discourse. *Computational Linguistics* 12(3):175-204, July-September, 1986.

[10] Hinrichs, E., D. Ayuso, and R. Scha. The Syntax and Semantics of the JANUS Semantic Interpretation Language. In R. Weischedel, D.Ayuso, A. Haas, E. Hinrichs, R. Scha, V. Shaked, D. Stallard (editors), *Research and Development in Natural Language Understanding as Part of the Strategic Computing Program*, chapter 3, pages 27-34. BBN Laboratories, Cambridge, Mass., 1987. Report No. 6522.

[11] Hinrichs, E.W., Ayuso, D.M., and Scha, R. The Syntax and Semantics of the JANUS Semantic Interpretation Language. In *Research and Development in Natural Language Understanding as Part of the Strategic Computing Program. Annual Technical Report December 1985 - December 1986*, pages 27-31. BBN Laboratories, Report No. 6522, 1987.

[12] Kaemmerer, W. and J. Larson. A graph-oriented knowledge representation and unification technique for automatically selecting and invoking software functions. In *Proceedings AAAI-86 Fifth National Conference on Artificial Intelligence*, pages 825-830. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc., 1986.

[13] MacLaughlin, D. *Parrot. The Janus Paraphraser.* BBN Report 7139, BBN Systems and Technologies Corp., Cambridge, MA, 1989.

[14] MacLaughlin, D. Koile, K., and Walker, E. *IRUS-II User's Manual.* BBN Report 7143, Bolt Beranek and Newman Inc., September, 1989.

[15] Meteer, M. *The Spokesman Natural Language Generation System.* BBN Report 7090, BBN Systems and Technologies Corp., Cambridge, MA, 1989.

[16] Moser, M.G. An Overview of NIKL, the New Implementation of KL-ONE. In Sidner, C. L., et al. (editors), *Research in Knowledge Representation for Natural Language Understanding - Annual Report, 1 September 1982 - 31 August 1983*, pages 7-26. BBN Laboratories Report No. 5421, 1983.

[17] Pavlin, J. and R. Bates. *SIMS: Single Interface to Multiple Systems.* Technical Report ISI/RR-88-200, University of Southern California Information Sciences Institute, February, 1988.

[18] Ramshaw, L. A. *Manual for SLS / RL (Release 1).* SLS Report 3, Bolt Beranek and Newman Inc., 1989.

[19] Reinhart, Tanya. *Anaphora and Semantic Interpretation.* Croom Helm, London and Sidney, 1983.

[20] Resnik, P. *Access to Multiple Underlying Systems in Janus.* BBN Report 7142, Bolt Beranek and Newman Inc., September, 1989.

[21] Scha, Remko J.H. Distributive, Collective and Cumulative Quantification. In J.A.G. Groenendijk, T.M.V. Janssen and M.B.J. Stokhof (editors), *Formal Methods in the Study of Language (Part 2)*, pages 483-512 Mathematisch Centrum, Amsterdam, 1981. Reprinted in: J.A.G. Groenendijk, T.M.V. Janssen and M.B.J. Stokhof (eds.): Truth, Interpretation and Information. Dordrecht: Foris, 1983.

[22] Scha, R. and Stallard, D. Multi-level Plurals and Distributivity. In *26th Annual Meeting of the Association for Computational Linguistics*, pages 17-24. Association for Computational Linguistics, June, 1988.

[23] Scha, R., Weischedel, R., and Ramshaw, L. Research and Development in Natural Language Processing in the Strategic Computing Program. *Computational Linguistics* 12(2):132-136, April-June, 1986.

[24] Sidner, C.L., Bates, M., Bobrow, R.J., Brachman, R.J., Cohen, P.R., Israel, D., Schmolze, J., Webber, B.L. and Woods, W.A. *Research in Knowledge Representation for Natural Language Understanding, Annual Report: 1 September 1980 - 3 August 1981.* BBN Report No. 4785, Bolt Beranek and Newman Inc., Cambridge, MA, 1981.

[25] Sidner, C.L., Bates, M., Bobrow, R., Goodman, B., Haas, A., Ingria, R., Israel, D., McAllester, D., Moser, M., Schmolze, J., Vilain, M. *Research in Knowledge Representation for Natural Language Understanding - Annual Report, 1 September 1982 - 31 August 1983.* Technical Report 5421, BBN Laboratories, Cambridge, MA, 1983.

[26] Sidner, C.L., Goodman, B., Haas, A., Moser, M., Stallard, D., Vilain, M. *Research in Knowledge Representation for Natural Language Understanding.* Technical Report 5694, BBN Laboratories, Cambridge, MA, September, 1984.

[27] Stallard, D. Data Modelling for Natural Language Access. In *The First Conference on Artificial Intelligence Applications*, pages 19-24. IEEE Computer Society, December, 1984.

[28] Stallard, D.G. A Terminological Simplification Transformation for Natural Language Question-Answering Systems. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 241-246. New York, June, 1986.

[29] Stallard, David. A Manual for the Logical Language of the BBN Spoken Language Sytem. July, 1988.

[30] Walker, E., Weischedel, R.M., and Ramshaw, L. IRUS/Janus Natural Language Interface Technology in the Strategic Computing Program. *Signal* 40(12):86-90, August, 1986.

[31] Webber, B.L. *A Formal Approach to Discourse Anaphora.* Technical Report 3761, Bolt, Beranek and Newman, Inc, 1978. Cambridge, Mass.

[32] Webber, Bonnie L. So What Can We Talk About Now? *Computational Models of Discourse.* MIT Press, 1983, pages 331-372.

[33] Weischedel, R.M., Bobrow, R., Ayuso, D.M., and Ramshaw, L. Portability in the Janus Natural Language Interface. In *Speech and Natural Language*, pages 112-117. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1989.

[34] Weischedel, R. M. A Hybrid Approach to Representation in the Janus Natural Language Processor. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 193-202. 1989.

[35] Woods, W.A. *Theoretical Studies in Natural Language Understanding, Annual Report.* BBN Report No. 4395, Bolt Beranek and Newman Inc. May, 1980.